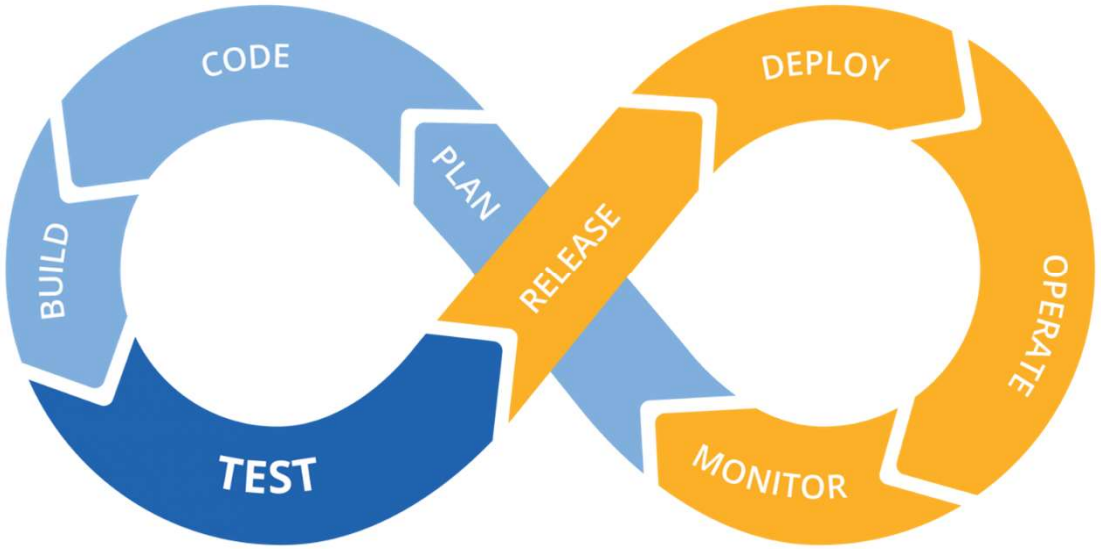
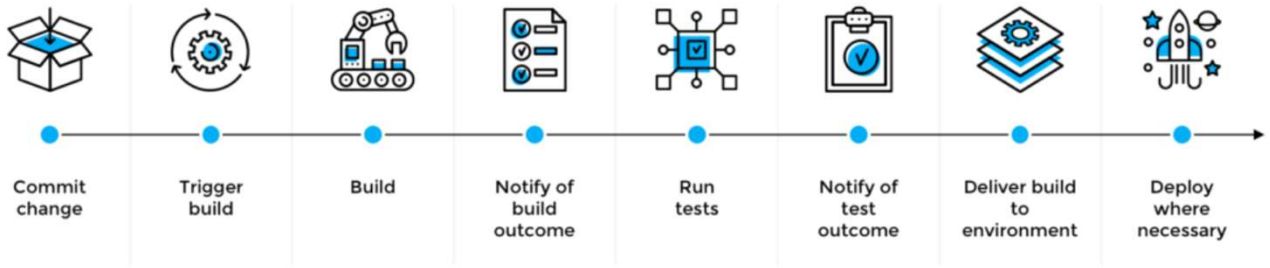


Ingeniería de Software III

Integración Continua y Despliegue Seguro



Contenido

- Cycle Time
- Entrega de Software
- Antipatrones de la entrega del software
- Release Candidate
- Principios de la entrega del software
- Integración Continua
- Despliegue y Entrega Continua
- Deployment Pipeline Pattern

Situaciones comunes

Necesito *que demuestres los nuevas "features" del software al cliente mañana*

Situación:

- Desarrolladores aún desarrollandolas
- El código compila
- Los unit tests pasan
- Pero va a llevar un par de días crear un release , crear un entorno de demo y finalmente instalarlo

¿No es poco razonable esperar una demo en un tiempo tan corto?

Situaciones comunes

*Hay un bug crítico en producción que está costando dinero al negocio, **necesito** un fix esta semana*

Situación:

- Sabemos que es un cambio rápido, una sola línea de código y un cambio en una tabla de la BD
- Pero el último release en PROD tomó un fin de semana entero trabajando hasta las 3 a.m. y la persona que lo hizo renunció después de eso
- Sabemos que el deployment se extenderá más allá del fin de semana esta vez

¿Por qué el negocio no entiende las complicaciones técnicas?

Cycle time (CT)

- ¿Cuánto tiempo le toma a una organización desplegar un cambio que involucra una sola línea de código?
- ¿Lo hace de una manera repetible y confiable?

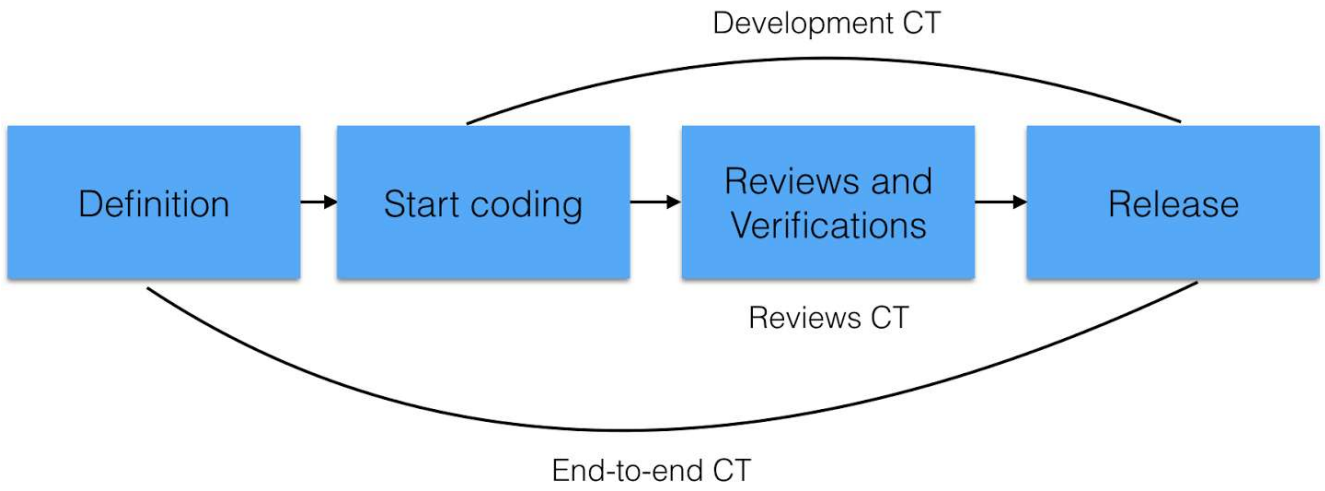
Cycle time (CT)

- Definición

  "El tiempo entre que se decide hacer un cambio y se tiene disponible en producción"

- A veces
  - Cycle time = meses o semanas
  - Cycle time = horas o minutos

- Clave para mejorar el CT → Automatización, buenas practicas y patrones



Entrega de software

- ¿Cómo entregar el software al cliente lo más rápido posible y de manera repetible y confiable?
- Las metodologías de software se centran en
  - requerimientos, desarrollo y la validación
- ¿Pero que pasa después?
  - The "last mile"
- Buenas Prácticas
  - CI / CD
- Patrón
  - Deployment pipeline

Antipatrones

✧Día de release

- Preparación manual de los entornos
  - Por un team de operations
- Instalación de dependencias
- App artifacts
  - Copiados a los servidores de producción
- Configuración
  - Copiada o creada manualmente
- La aplicación es iniciada pieza por pieza

Muchas cosas pueden salir mal

Antipatrones - Despliegue manual

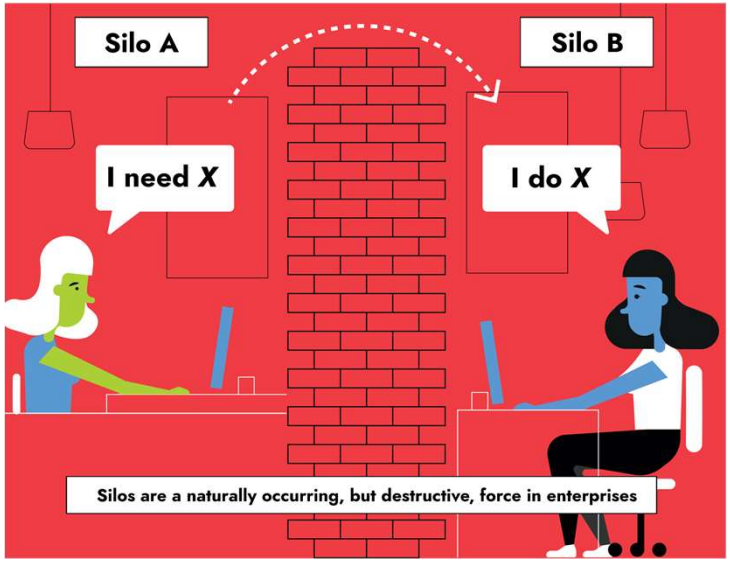
- El despliegue se hace de manera manual
- Generalmente por un team de operations

Antipatrones - Despliegue manual

- Signos
  - ✧Documentación extensa
  - ✧Verificación manual
  - ✧Interacción constante con desarrolladores
  - ✧Correcciónes al proceso de release mientras se está ejecutando
  - ✧Diferencias entre los environments
  - ✧Releases que toman más que minutos
  - ✧Resultado impredecible
  - ✧Ojos rojos frente al monitor a las 2 a.m. tratando que las cosas funcionen

Antipatrones - Despliegue manual

- Problemas de los procesos no automatizados
  - Errores cada vez que se ejecutan
  - No son repetibles, ni confiables
  - Ineficientes, debugging, etc
  - Necesitan documentación, una tarea que lleva tiempo y la doc debe mantenerse
  - Depende de un experto en instalar el software
  - A veces se dice que un proceso manual es “más auditable” -> FALSO

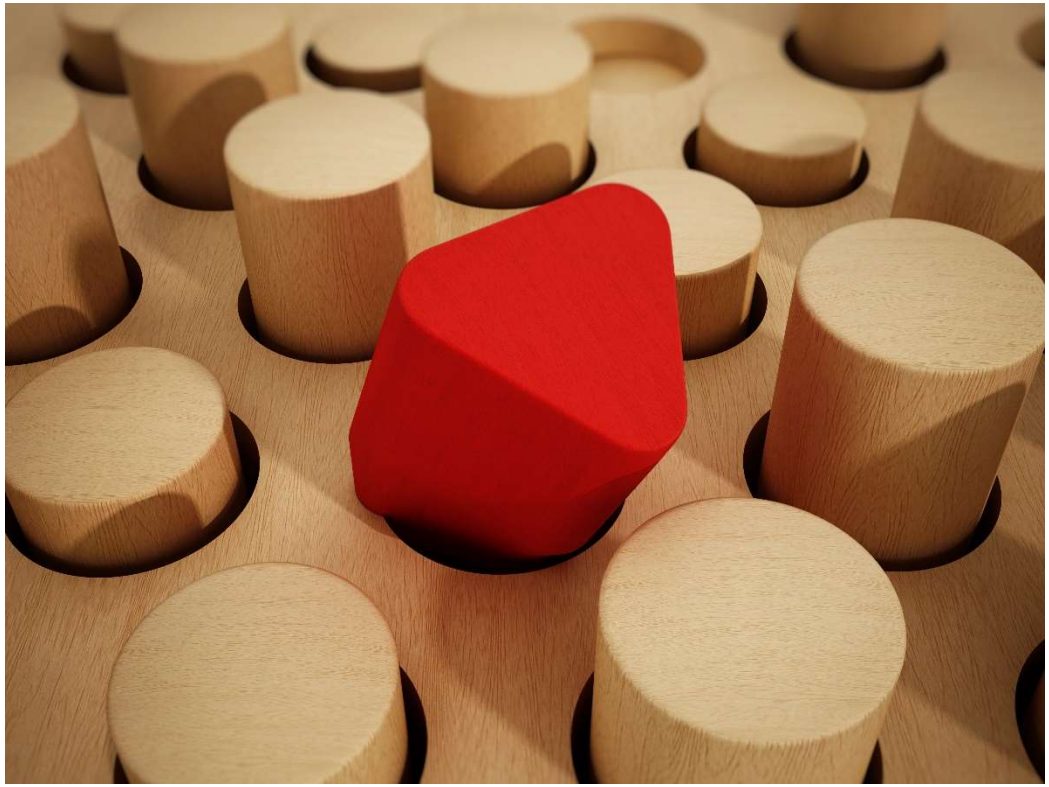


Antipatrones - Despliegue manual

- Alternativa
  - Automatización
    - Sólo 2 tareas, seleccionar la version a desplegar y presionar el botón "desplegar“

Antipatrones - No testear en un entorno como producción

- El software se despliega en un entorno como producción solo cuando el desarrollo ha terminado, no antes



Antipatrones - No testear en un entorno como producción

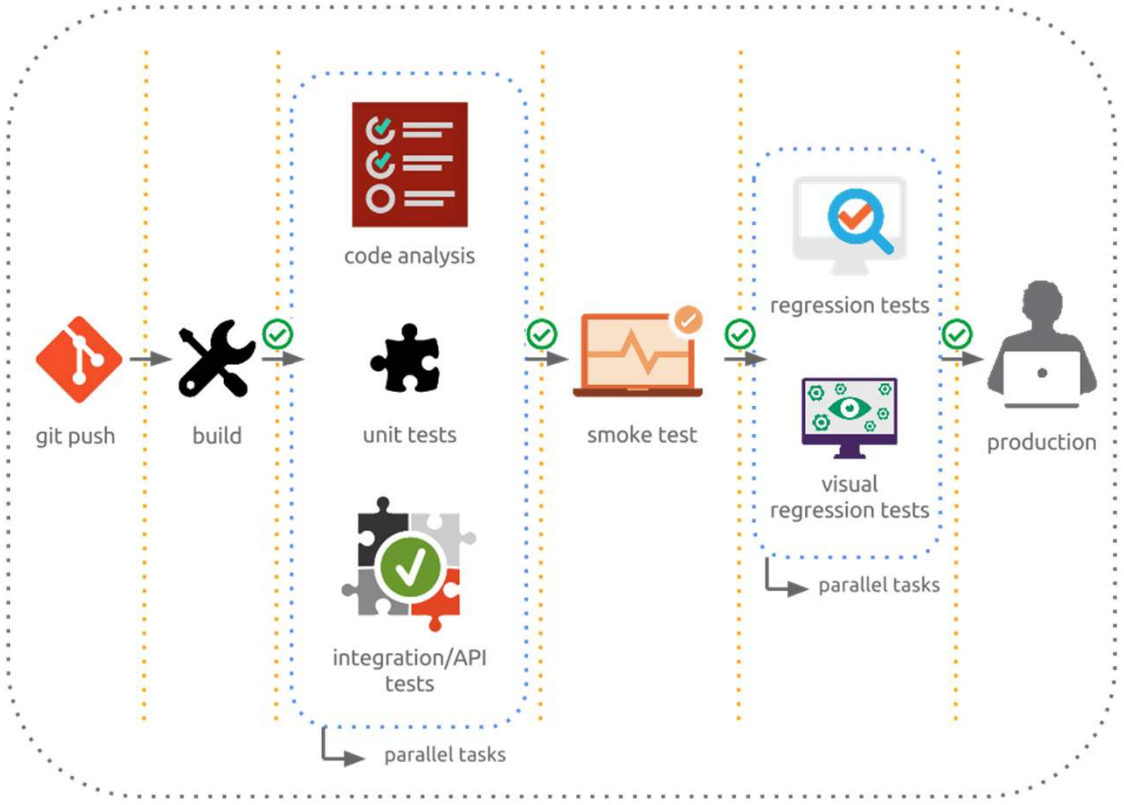
- Signos
  - Si se testeó fue en entornos de desarrollo
  - Release a staging
    - Operations interactúa con el software por primera vez
    - team específico para releases a staging y a producción
  - Production environment
    - Es muy caro,
    - Está muy controlado,
    - Nadie se preocupó por crear un env de testing similar al de prod
  - Desarrollo prepara
    - instaladores, configuración, etc
    - para operations
    - sin probar en un entorno como el de prod
  - Desarrolladores y Operations no colaboran hasta el día del release

Antipatrones - No testear en un entorno como producción

- Problemas
  - El primer release es generalmente el más problemático
  - Mientrás > sea Release cycle
    - + tiempo se sostendrán assumptios
    - + tiempo tomará arreglarlas
  - Costo de coordinación entre silos
    - Enorme,
    - Infierno de “tickets”
  - Mientras + diferencias entre dev y prod
    - + y > assumptions
    - + probabilidad de encontrar problemas

Antipatrones - No testear en un entorno como producción

- Alternativa
  - Testing, deployment y release parte del desarrollo
  - Integración continua y deployment continuo para testear software y deployment



Antipatrones - Administración manual de la configuración en Producción

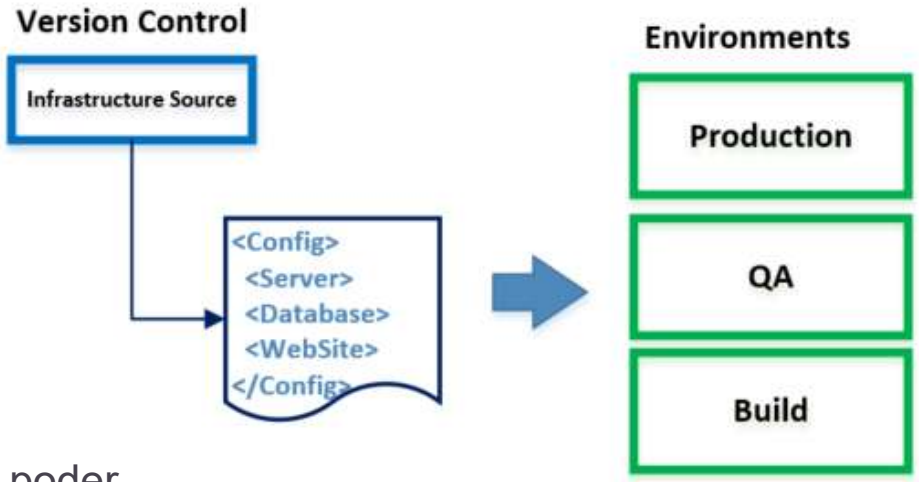
- Team específico para cambiar configuración en producción
- Cambio manual
- Cambio directo en prod

Antipatrones - Administración manual de la configuración en Producción

- Signos
  - Despliegue en staging 
    - Pero despliegue en producción 
  - Operations demora en preparar el entorno
  - No se puede hacer rollback del cambio
  - Ej. Comportamiento diferentes de nodos iguales en un cluster
  - Configuraciones se hacen directamente en producción

Antipatrones - Administración manual de la configuración en Producción

- Alternativa
  - Toda la conf
    - En VCS
    - Aplicada automaticamente desde el VCS
  - Recreación de la infraestructura
    - Automatizada
    - Repetitiba
  - Cambios prod
    - Registrados
    - Auditables
  - El mismo sistema automatizado debe poder hacer rollback
  - IaC -> Infrastructure as Code



¿Se puede mejorar?

✧Día de release

✧Actividades de release

- ✧“aburridas”,
- ✧incluso en entornos complejos
- ✧Bajo riesgo
- ✧Bajo costo
- ✧Rápidas
- ✧Repetibles
- ✧Predecibles

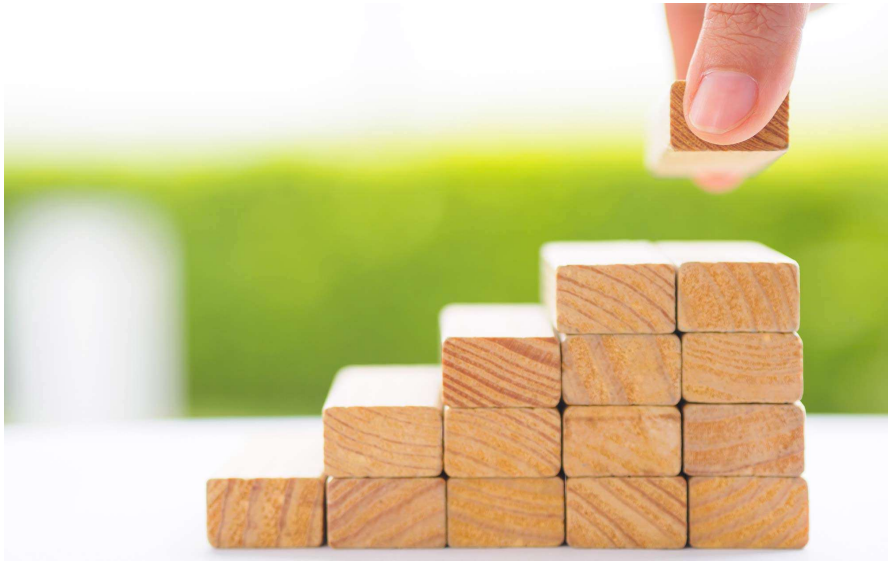
¿Cómo lo logramos?

- Goal
  - Velocidad de entrega (reducir el cycle time)
    - Hay un costo de oportunidad asociado a la no entrega o entrega demorada del software
    - Permite verificar la utilidad de la feature o bug fix que se entrega
    - Minimizar el ciclo de entrega acelera el feedback
    - Permite basarse en el conocimiento
  - Quality
    - El software debería servir a su propósito
    - No se trata de perfección
    - Debe mantener niveles apropiados de calidad

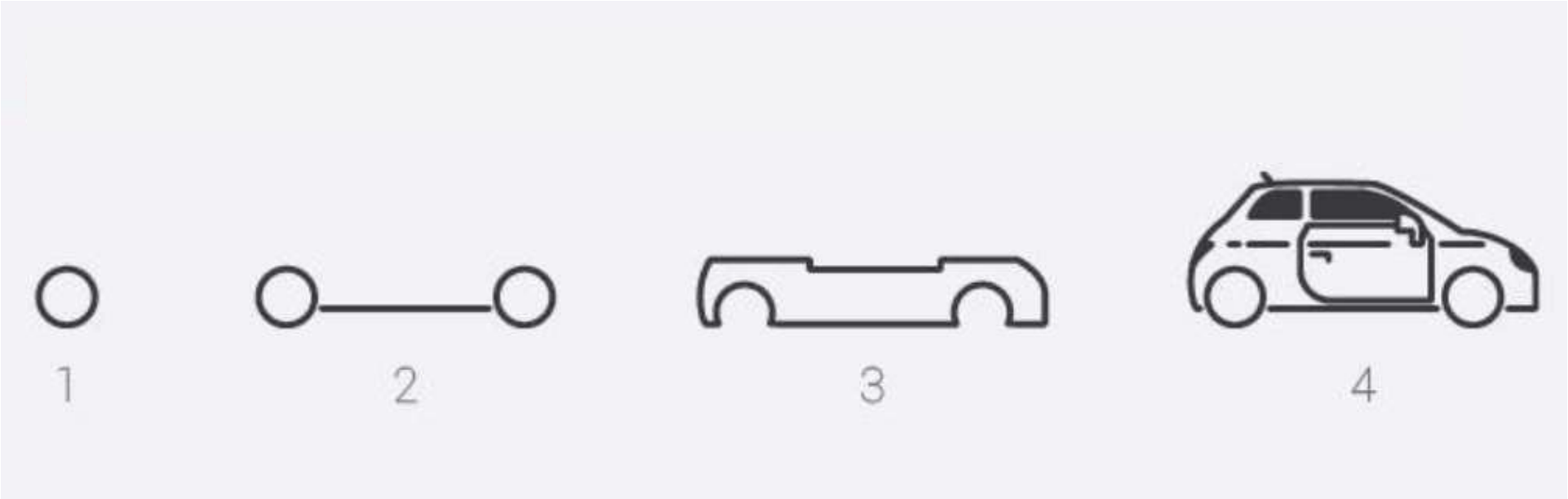
Queremos encontrar maneras de entregar software de calidad, que le sume valor al cliente, de una manera eficiente, rápida y confiable.

¿Cómo lo logramos?

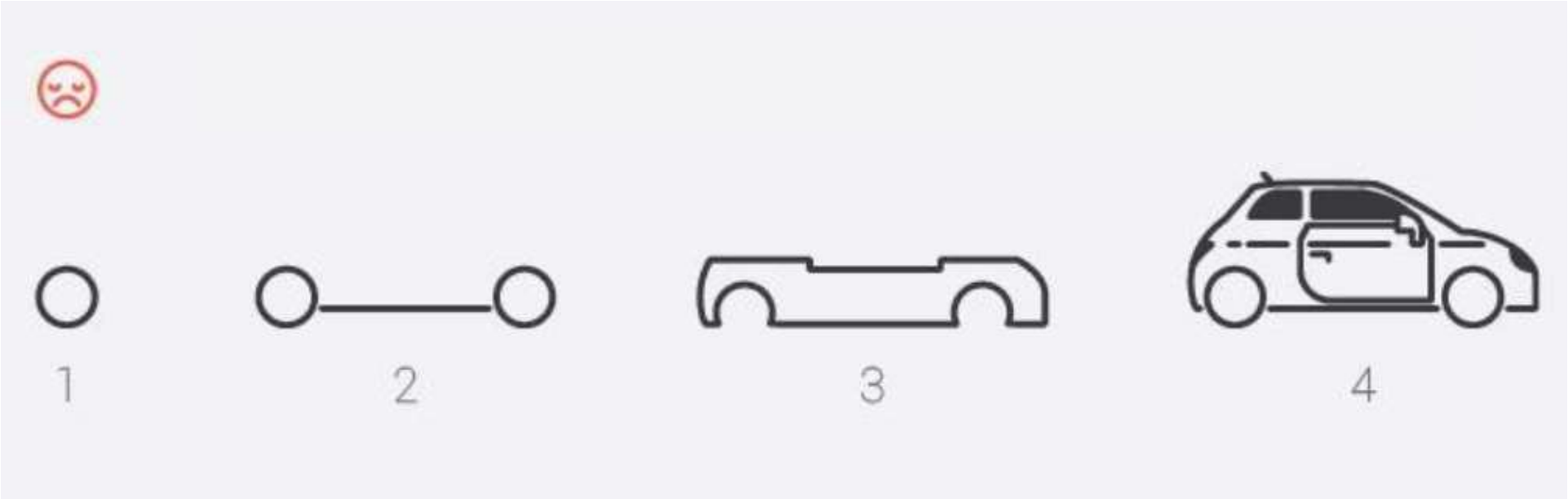
- Ciclos cortos y alta calidad, ¿cómo?
  - Automatizado
    - Repetible.
    - No sea error-prone.
    - Control sobre el proceso de release
    - Arte ✗
    - Disciplina de ingeniería ✓
  - Frecuente
    - Incrementos pequeños
    - Reducir el riesgo
    - Feedback frecuente



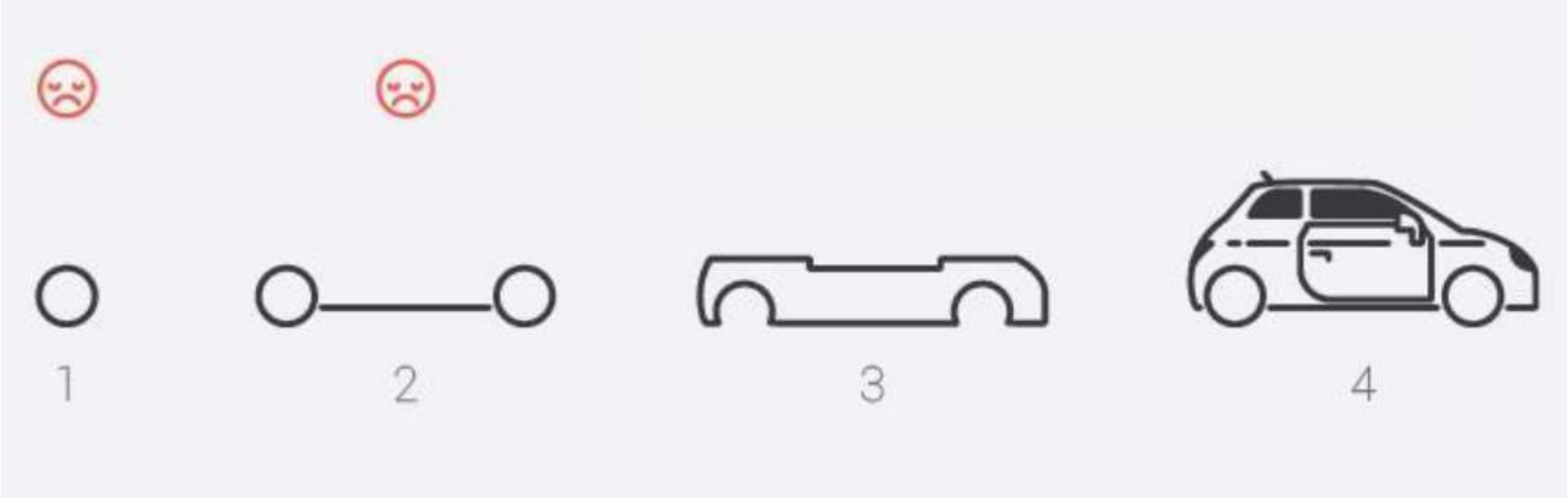
Iterative



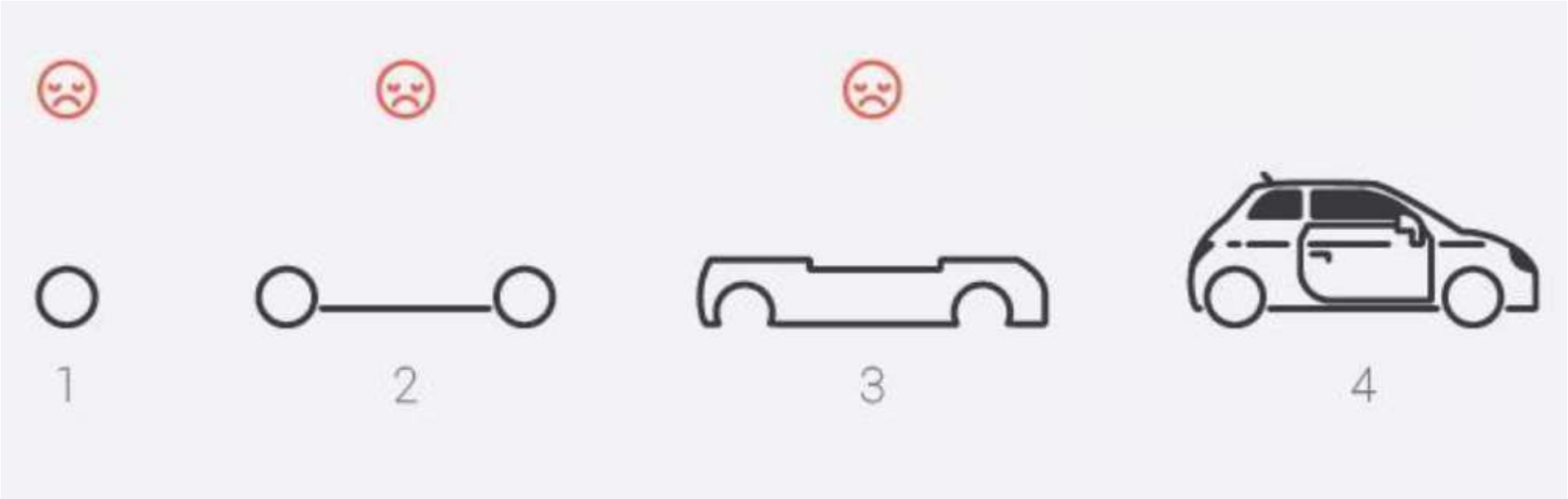
Iterative



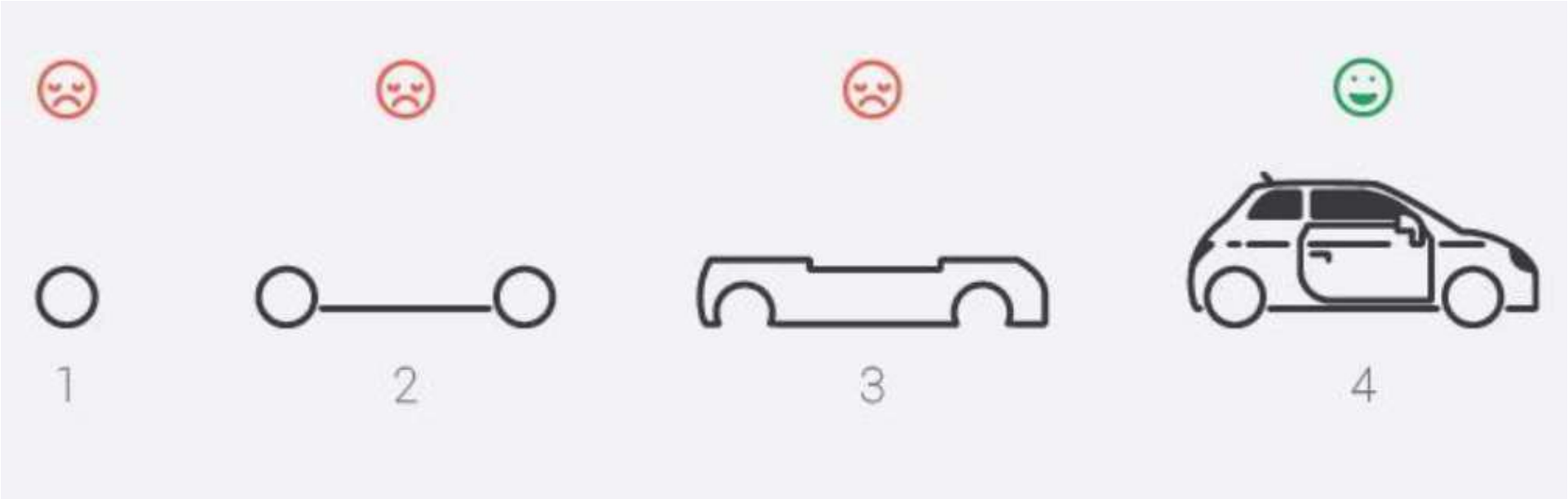
Iterative



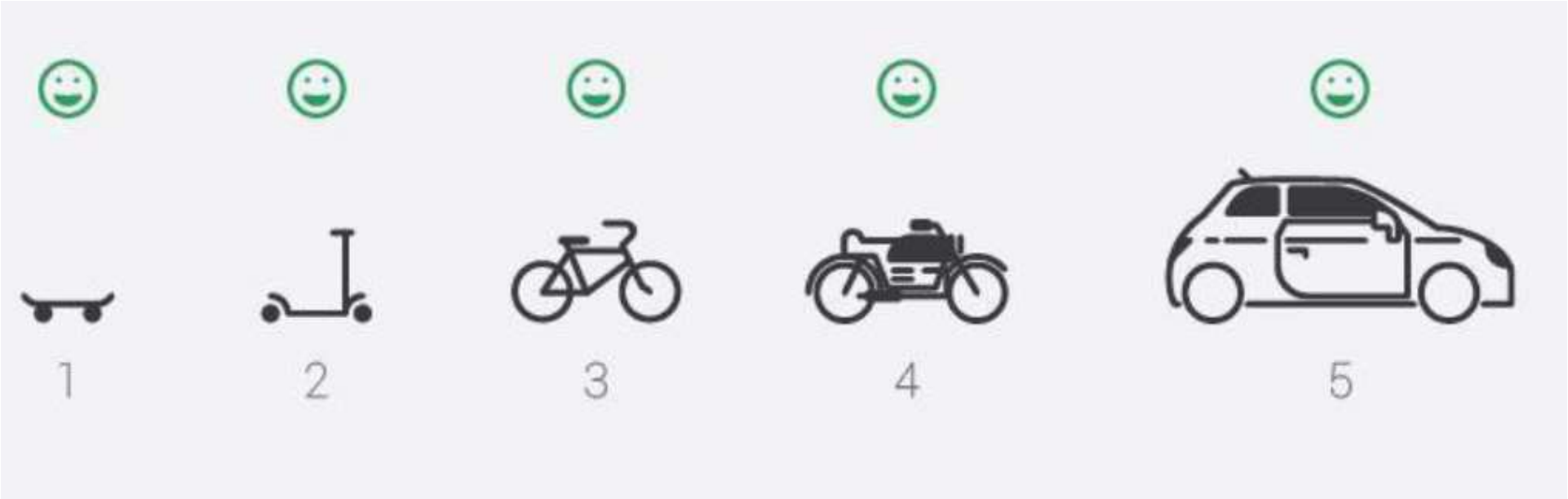
Iterative



Iterative



Iterative



¿Cómo lo logramos?

- Feedback es esencial
- Hay 3 criterios para que el feedback sea útil
  - Todo cambio debe disparar el proceso de feedback
  - El feedback debe ser logrado lo más rápido posible
  - El team de desarrollo debe actuar sobre el feedback recibido

Todo cambio debe disparar el proceso de feedback

- Código fuente
  - Cambia el ejecutable
  - Binario -> construido y testeado
  - Automatizados
  - Integración continua -> CI

- Configuración
  - Binario debe ser instalado en los entornos, testing, producción, etc
  - Binarios se reusan, nunca se re-construye
  - Cambio entre entornos -> capturado como configuración
  - Todo cambio de conf (en cualquier environment) -> testear

- Entorno
  - Cambios de entorno -> testear
  - Ejemplos:
    - Sistema operativo,
    - Software stack que soporta la aplicación,
    - Configuración de la red,
    - Infraestructura
    - Sistemas externos

Todo cambio debe disparar el proceso de feedback

- El proceso de feedback involucra testear cada cambio de una manera automatizada

- Tipo de checkeos
  - El proceso de construcción del sistema debe funcionar.
    - **La sintaxis es correcta**
  - Los unit tests deben pasar.
    - **El código se comporta como se espera**
    - Criterios de calidad
      - test coverage
  - Los tests funcionales deben pasar.
    - **El sistema logra el criterio de aceptación del negocio**
  - Los test no funcionales deben pasar.
    - **El sistema se comporta como se espera en términos de performance, escalabilidad, etc**
  - El software se debe explorar manualmente y revisar con el usuario.

El feedback debe ser recibido lo antes posible

- ✧Feedback rápido => Automatización

- ✧Tareas interesantes => Personas
- ✧Tareas repetitivas => Máquinas

- ✧Deployment Pipeline requiere muchos recursos

- ✧Se pueden tener diferentes set de tests
  - ✧Commit-stage
    - ✧Tests de ejecución rápida
    - ✧Que sean independientes del environment
    - ✧Tener un test coverage alto
    - ✧Si estos tests fallan no se puede relesear el software

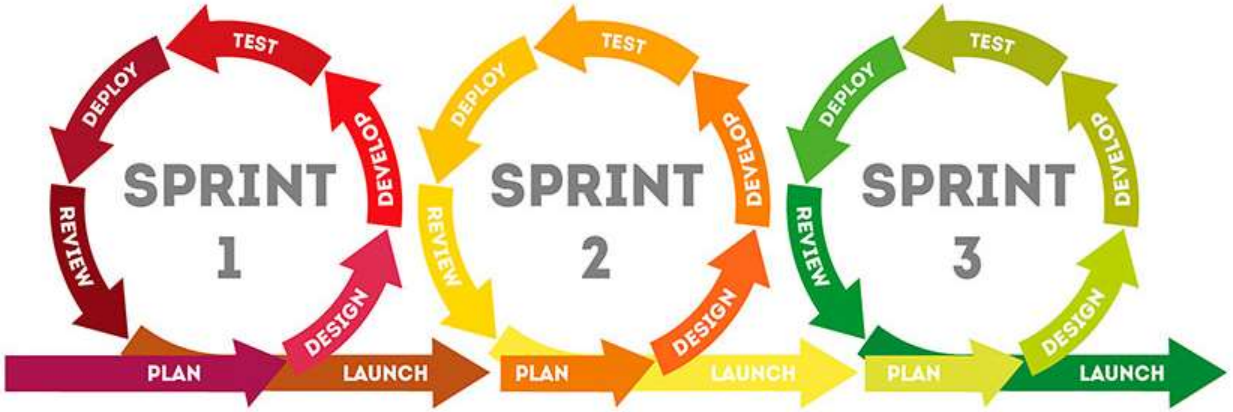
  - ✧Tests funcionales
    - ✧Son más lentos
    - ✧Si fallan aún se puede elegir relesear el software con known bugs
    - ✧Deben correr en un entorno lo más parecido a producción posible

  - ✧Tests no funcionales
    - ✧Requieren más tiempo
    - ✧Requieren más recursos

- ✧Lograr el feedback lo más rápido posible

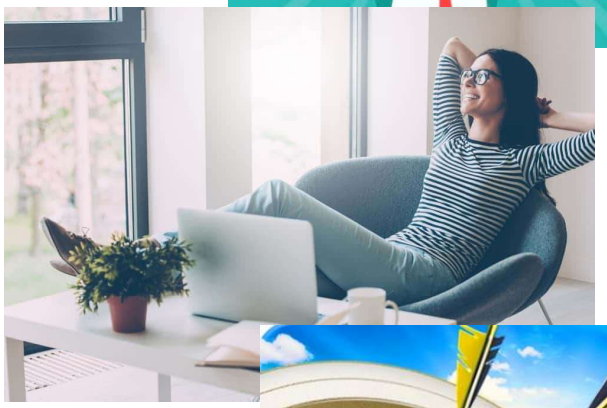
El team de desarrollo debe recibir feedback y luego actuar

- El feedback debe llegar a todos los stakeholders
- Un deployment pipeline
  - Feedback inmediato
  - Feedback visible
- Se debe reaccionar ante el feeback,
  - Es necesario un marco de trabajo ágil



Beneficios de este enfoque

- ✧Proceso de release, repetible, confiable y predecible
- ✧Reducción del ciclo de release y aumento de la calidad
- ✧Le da más poder al team
  - ✧Se puede elegir una version deseada y deployarla en un entorno deseado
  - ✧No hay que esperar un “buen release”
- ✧Reduce errores
  - ✧Provenientes de una pobre Administración de la Configuración
- ✧Reduce el stress
  - ✧Porque las tareas se hacen triviales, repetibles y predecibles
- ✧Flexibilidad de despliegue
  - ✧Se vuelve trivial iniciar la app en un nuevo entorno
- ✧La práctica hace a la perfección
  - ✧Usar el mismo mecanismo de deployment en todos lados



Release candidate

- Identificar builds que son potencialmentes releseables
- Tradicionalmente
  - Release candidates detectados al final del proceso de Desarrollo

  - Proceso manual y costoso de testeo para designar un release como candidato

Release candidate

- Demorar el testeo, va en detrimento de la calidad

- Los defectos son menos costosos de arreglar mientras se encuentren más cerca de dónde fueron introducidos
  - Costo de calidad pobre

- **Cada check in debería ser un release candidate**

- ¿Cómo?
  - Automatizar agresivamente,
  - Tests suites apropiadas

Principios de la entrega de software

- Prácticas
- Guías
- Comunes a los procesos de entrega de software efectivos

Principios de la entrega de software

- **Crear procesos de release repetibles y confiables**
  - Debería ser sencillo relesear software
  - Porque lo has hecho ciento de veces antes
  - Tan simple como presionar un botón
  - Deriva de los dos principios que siguen
    - Automatizar
    - Mantener todo bajo control de la configuración

Principios de la entrega de software

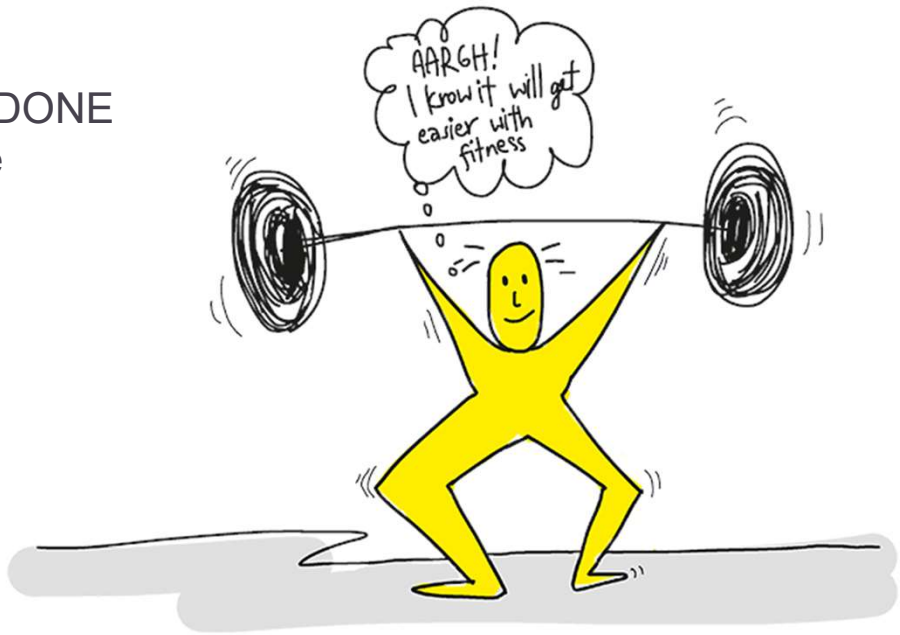
- **Automatizando casi todo**
  - Desplegar involucra
    - Provisioning del entorno
    - Instalación de la versión correcta de la aplicación
    - Configuración de la aplicación
  - Todo lo anterior => automatizado
  - Validaciones automatizadas
  - Cosas no pueden automatizarse
    - Exploración manual
    - Approvals
  - Excusa => es más fácil hacerlo manual

Principios de la entrega de software

- **Mantener todo bajo control de la configuración**
  - TODO
    - No solo código fuente
  - Un nuevo desarrollador
    - Clona el repositorio
    - Con un comando buildea
    - Con un comando deploya en cualquier entorno
      - Incluyendo su máquina
  - Builds identificables
    - Unívocamente
    - De manera completa

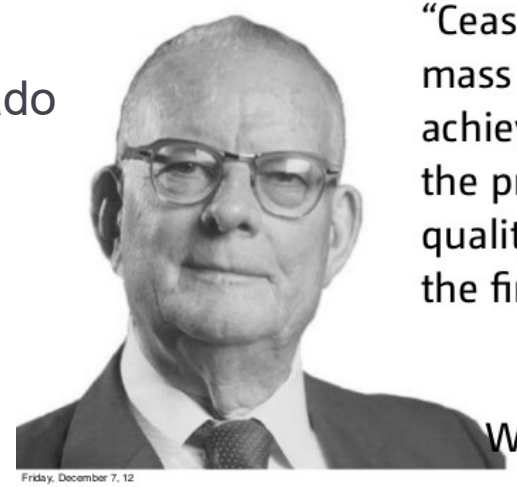
Principios de la entrega de software

- **Si "duele", hazlo más frecuentemente y trae el dolor antes**
  - Si releaser el software es "doloroso"
    - hazlo frecuentemente,
    - a entornos production-like
  - Si crear documentación es doloroso,
    - Hazlo como parte del proceso ,
    - No lo dejes para el último,
    - Que sea parte de la definición de DONE
    - Automatizada cuanto sea possible
  - Integración es doloroso
    - Hazlo para cada cambio
  - Testing es doloroso
    - Hazlo de manera continua
    - No lo dejes para el último



Principios de la entrega de software

- **Build quality in**
  - Las inspecciones son demasiado tarde, la calidad buena o mala ya está en el producto
  - Las inspecciones agregan costo pero no valor
  - Los defectos deben encontrarse lo más pronto posible, idealmente en la misma etapa que fueron introducidos
  - Integración continua, test automatizado
  - Testing NO es una fase, es continuo

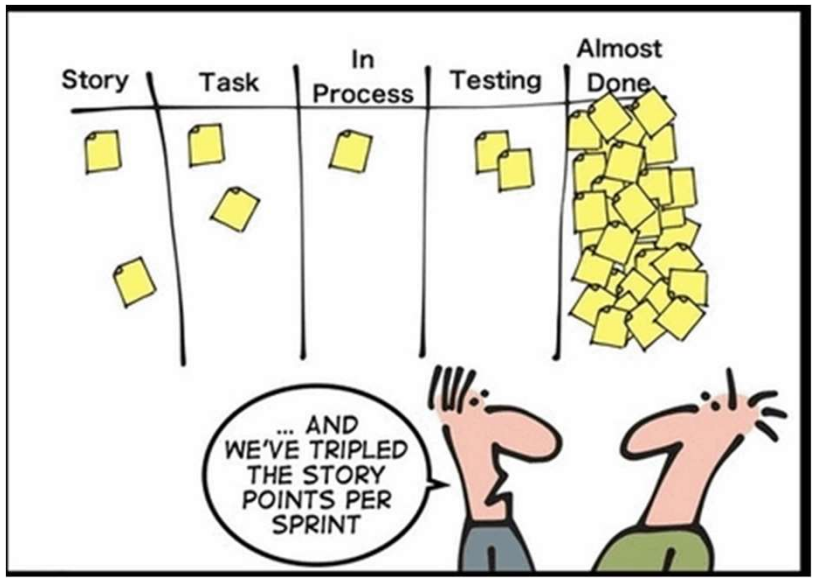


"Cease dependence on mass inspection to achieve quality. Improve the process and build quality into the product in the first place"

W. Edwards Deming

Principios de la entrega de software

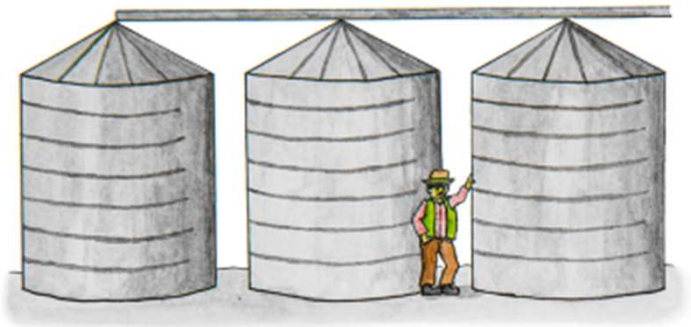
- **DONE significa RELEASED**
  - Muchas veces done para el desarrollador no es DONE
  - DONE debería ser cuando la feature le está agregando valor al usuario
  - Inicialmente, DONE => demostrado exitosamente
    - prod-like env
  - Idealmente, DONE => en manos del usuario

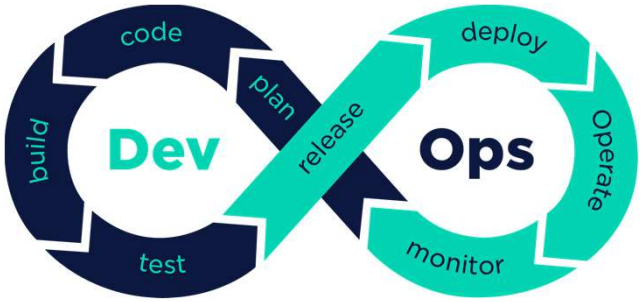


- Done involucre a todo el team!

Principios de la entrega de software

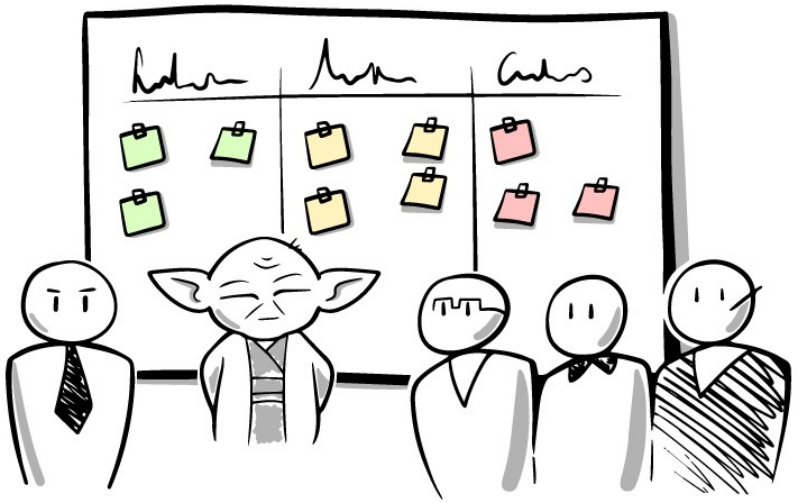
- **Todos son responsables del proceso de release**
  - El team es responsable del éxito o fracaso
  - A veces
    - devs => testers => operations
    - La culpa se pasa de unos a otros
  - Todos deberían estar involucrados en el proceso de entrega
  - Comunicación es esencial
    - Desde el comienzo del Proyecto y de manera regular
  - Demo
    - En un lugar donde todos puedan verlo





Principios de la entrega de software

- **Mejora continua**
  - Software mejora iterativamente a lo largo del desarrollo
    - El proceso de entrega debe mejorar también
  - Hacer retrospectivas acerca del proceso de entrega
    - Asignar actions, medir luego de aplicarlas
    - Deming cycle: plan, do, study, act
  - Toda la organización
    - enfocada => mejora continua



Entrega de software - Pattern

- Deployment pipeline

  "Manifestación automatizada del proceso de build, deploy, test y release de la aplicación"

- Proceso visible para todos
  - Fomenta la colaboración
- Retroalimentación inmediata
  - Problemas identificados y resueltos tempranamente
- Permite el despliegue a voluntad
- Evolución lógica de de CI -> **CI/CD**

UNIVERSIDAD CATÓLICA DE CÓRDOBA
Universidad Jesuita

Continuous Integration

Proyectos sin integración continua

- Hay un estado “extraño” pero no poco común en los proyectos de software
  - La aplicación está en un estado en el que no funciona aún
- Mayormente en proyectos desarrollados por teams grandes
  - A nadie le interesa tratar de correr la aplicación como un todo hasta que esté terminada
- *En proyectos con long-lived branches*
- En proyectos que los tests de aceptación se dejan para el final
- Se suele planear fases de integración largas, muchas veces porque no se sabe cuanto puede tardar

Integración Continua

✧Por otro lado están los proyectos que usan integración continua

✧Descrito 1ero => 1999 por Kent Beck en Xtreme Programing

✧Un cambio de paradigma

- ✧Sin CI, software"roto" hasta que alguien prueba lo contrario: durante la fase de testing o integración
- ✧Con CI: se verifica que el software funciona con cada cambio, y si se rompe se arregla inmediatamente

✧Objectivo de CI

- ✧"tener la aplicación en un estado funcional constantemente

✧Menos bugs, releases más rápidos

✧Práctica esencial

Integración Continua

- Antes de comenzar:
  - Version Control
    - Todo tiene que estar en un sistema de control de versions

  - Builds automatizados
    - Los scripts de build tiene que ser tratado como código base del producto

  - Compromiso del equipo de desarrollo
    - Es una práctica, no una tool
    - Requiere disciplina por parte del equipo
    - Hacer checkins frecuentes y pequeños
    - Compromiso de arreglar el build apenas se rompe

Integración Continua

✧Pasos, una vez que el cambio esté listo para checkin

1. Fijarse si el build está corriendo
   - Si falla, arreglarlo con el resto del team
2. Cuando el build y los tests pasan
   - Actualizar el code en el local env
3. Ejecutar el build y los test cases localmente
4. Si el build y los test pasan localmente, hacer checkin
5. Esperar que el build server buildee con nuestros cambios
6. Si falla, arreglar el problema inmediatamente – Volver al paso 3
7. Si el build pasa seguir con la próxima tarea ☺

Integración Continua

✧Pre-requisitos:

✧Hacer checkin regularmente
- ✧A trunk o main o master
- ✧Al menos 2 veces al día
- ✧Origina cambios más pequeños y es menos probable que rompan el build
- ✧Se tiene una version reciente que funciona a la cual revertir los cambios

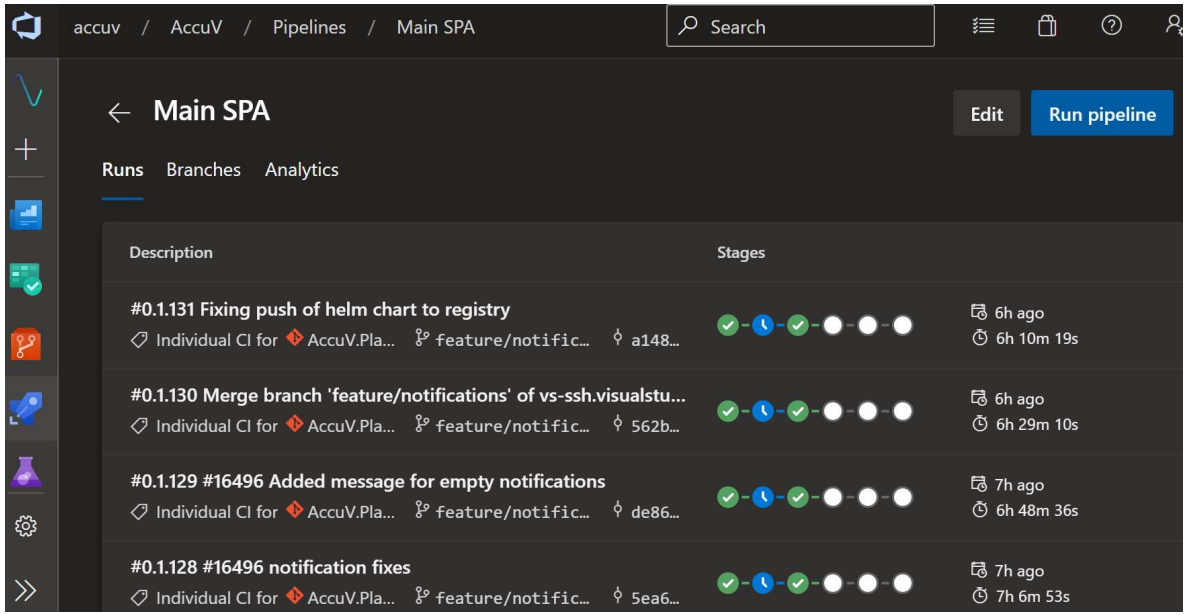
✧Crear un test suite exhaustivo
- ✧Unit tests
- ✧Component tests
- ✧Acceptance tests

✧Mantener el build y el proceso de testing corto
- ✧El límite es 10 minutos
- ✧5 minutos es mejor
- ✧Parece contradecir el anterior, diferente tipos de tests en diferentes etapas -> pipeline

✧Usar workspaces privados
- ✧Los desarrolladores debe poder compilar y correr los tests en sus entornos privados
- ✧Entornos de desarrollo compartidos deberían ser la excepción

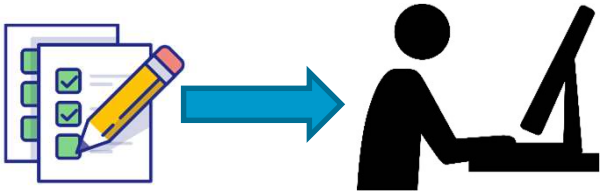
Integración Continua

- Operación Básica:
  - Esencialmente hay 2 components
    - Un servicio que ejecuta un “workflow” a intervalos regulares
    - Un servicio que provee una vista de los resultados del build y los tests, notifica el estado de los builds, provee acceso a los instaladores, y reportes



Integración Continua

- Prácticas esenciales
  - No hacer commit en un build roto 
  - Corre los tests localmente antes de hacer commit o usar al building server para que lo haga
    - Update workspace
    - Run local build
    - Run local tests
    - Commit if success



  - Esperar que termine el build en main después del commit antes de continuar
    - Hay que asegurarse que pase y sino arreglarlo o revertir el cambio

Integración Continua

- Prácticas esenciales
  - No irse a casa con un build roto
    - Para evitar irse tarde hacer check ins frecuentes y más temprano, no a último momento

  - Siempre estar preparado para revertir a la version anterior
    - Si no podemos arreglar el problema por cualquier razón hay que revertir el cambio

Integración Continua

✧Prácticas esenciales

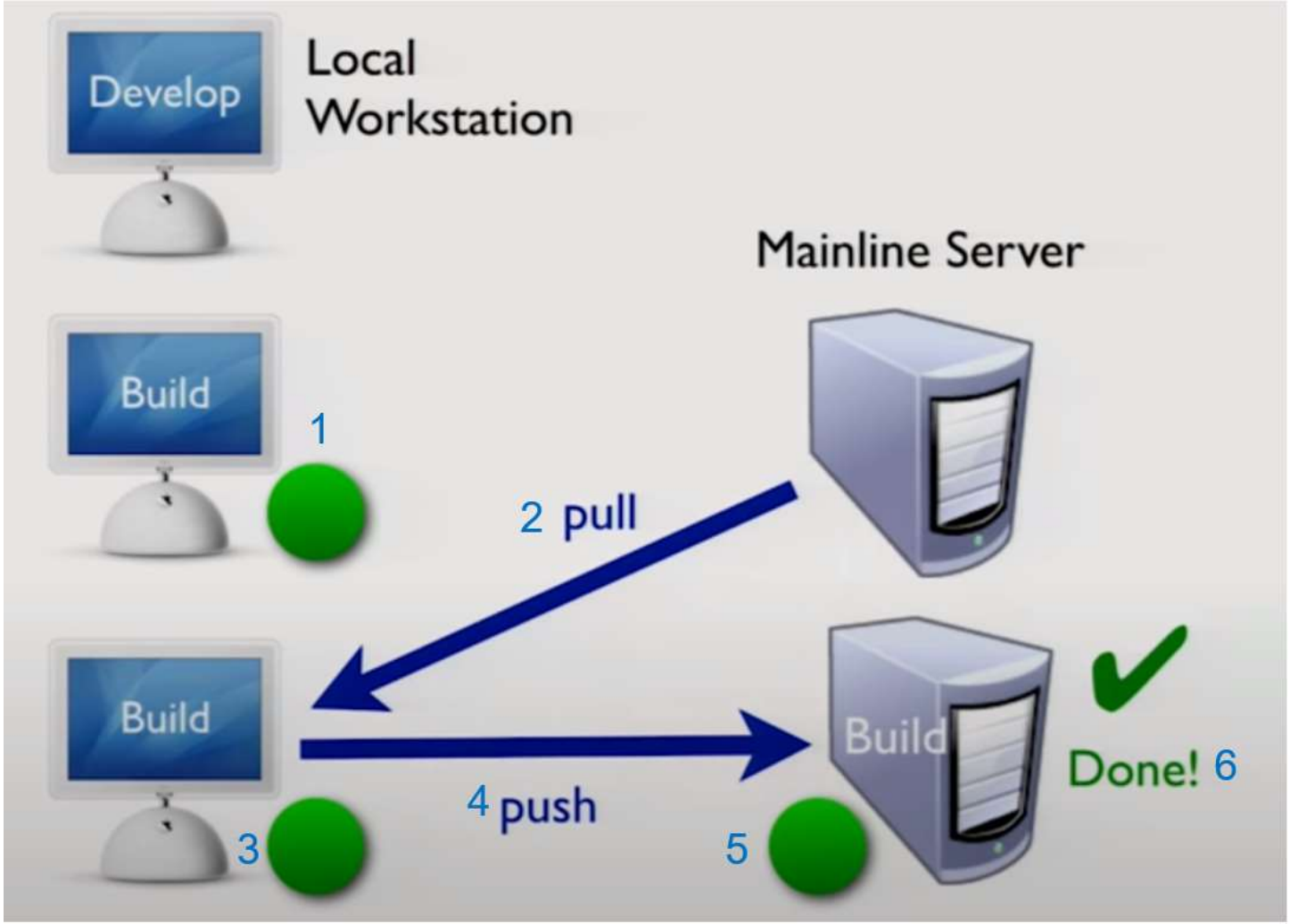
✧Time-box fixing antes de revertir el cambio

✧No comentar tests cases que fallan

✧Aceptar la responsabilidad de todas las roturas por nuestros cambios

✧Test-Driven development

Continuous Integration



Deployment Pipeline

Introducción

- CI
  - Aumenta productividad y calidad
  - Incrementa el trabajo en equipo
  - Feedback rápido del código
  - Asegura que el código compila y los UT pasan
  - Pero no es suficiente
- Ci
  - Se enfoca en el equipo de Desarrollo
  - La salida es normalmente la entrada para el equipo de testing y para el resto del proceso de release

Introducción

- Mucho Desperdicio -> proceso de software -> software a través de testing y Operations
  - Operations esperando
    - documentación
    - fixes
  - Testers esperando
    - “Buenos” builds para testear
    - La creación de “Buenos” entornos
    - La instalación del software

  - Desarolladores
    - Recibiendo reportes de errores mucho después de haber empezado una nueva funcionaldiad
    - Descubriendo al final del proceso que la arquitectura no es la adecuada

Introducción

- Esto lleva a un software
  - Imposible de instalar
    - Demoró demasiado en llegar a un entorno como el de producción
  - Defectuosos (buggy)
    - Ciclos de feedback entre dev, test y ops es demasiado largo

How to improve?

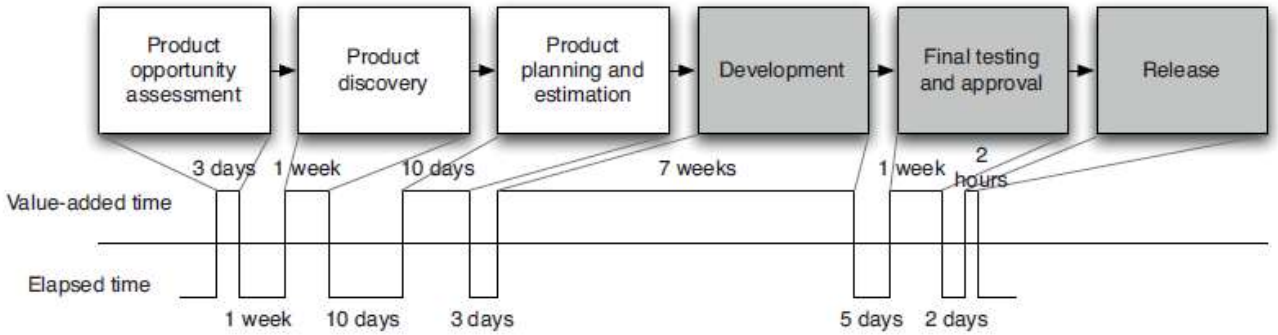
- Algunas mejoras
  - Devs escribir código para producción
  - CI en entornos como el de producción
  - Crear cross-functional teams

- Solución crear un enfoque end-to-end
  - Automatizar todo el proceso
    - Build, deploy, test y reléase
  - Esto crea un bucle de retroalimentación poderoso dado que es fácil deployar
    - En test envs -> se obtiene feedback rápido del código y el proceso de deploy

Pull system

- On demand, presionando un botón
  - Testers -> deployan a testing env
  - Operations -> deployan a staging o production
  - Devs -> ven qué builds están en qué stage y cuáles son los problemas encontrados
  - Managers -> métricas (cycle time, throughput, quality, etc)
- Todos tienen acceso a lo que necesitan y visibilidad del proceso entero
  - Permite mejorar el proceso, identificando y eliminando bottlenecks
  - Proceso más rápido y mejor

Definición



- Deployment pipeline

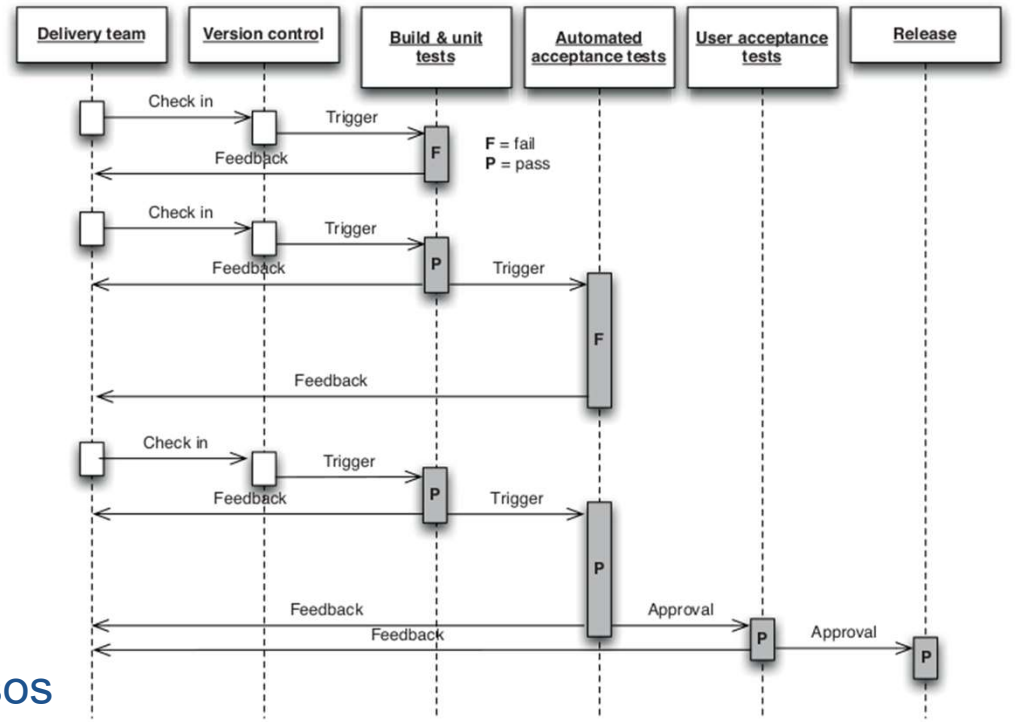
  "Manifestación automatizada del proceso que se sigue para llevar el código desde el VCS a manos de los usuarios"

- Involucra
  - Construir (build) el software
  - Transitar varias etapas de testeo y deployment
  - Requiere colaboración entre varios individuos y equipos
  - Deployment pipeline modela este proceso

- Deployment pipeline -> última parte del stream value

Deployment pipeline

- Entrada -> Una versión de VCS
- El proceso lleva al build a través de una serie de etapas
- En cada etapa se evalua al build desde diferentes perspectivas
- más lejos llega un build, mayor confianza en la calidad del build
- -> mayor la cantidad de recursos invertidos en ese build
- -> Env más parecido a prod



- Objetivo: eliminar builds defectuosos
  - Lo más rápido posible
  - Para obtener feedback rápido
  - Builds defectuosos no continúan en el pipeline

Deployment pipeline

- -> mayor la cantidad de recursos invertidos en ese build
- -> Env más parecido a prod

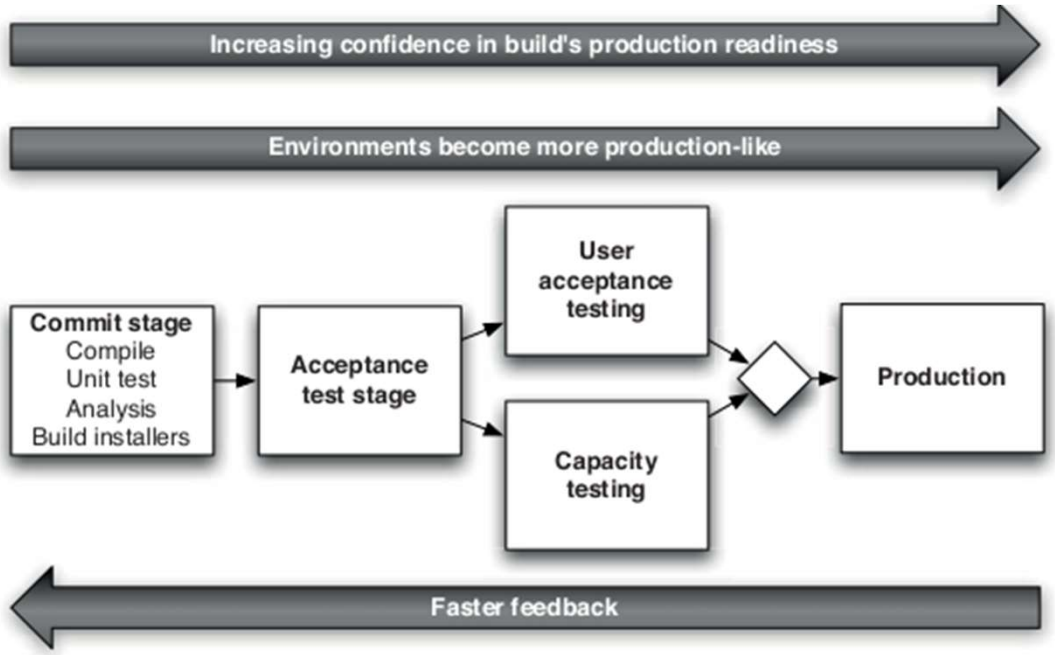


Figure 5.3 *Trade-offs in the deployment pipeline*

Consecuencias

- Primero
  - No se deployan en prod releases que no fueron “ensayados”
  - Se evitan bugs de regresión, incluso en fixes de emergencia
  - Se mitigan riesgos relacionados al entorno de ejecución
- Segundo
  - Releases, rápidos, confiables y repetibles
  - Se pueden hacer releases más frecuentes
  - Se puede volver atrás o avanzar si se lo desea
  - Los releases no tienen riesgo

¿Cómo?

- Para lograr este estado “envidiable”
  - Automatizar un conjunto de tests que prueben que nuestro reléase candidate cumple con su propósito
  - Automatizar deployment a todos los entornos, test, staging, producción
  - Remover toda tarea manual
  - Las etapas que se deben automatizar dependen de cada proceso, pero se pueden identificar algunas comunes

Deployment pipeline - Estructura

- Commit stage
  - Verifica que el sistema trabaja a nivel técnico
  - Compila, pase el suite automatizado de tests cases (UT)
  - Ejecuta analisis estático del código
  - Construye instaladores

- Automated acceptance
  - Verifica que el sistema funciona a nivel funcional
  - El comportamiento cumple con las especificaciones del cliente

Deployment pipeline - Estructura

- Otros tests
  - Tests no funcionales
  - de capacidad
  - de stress

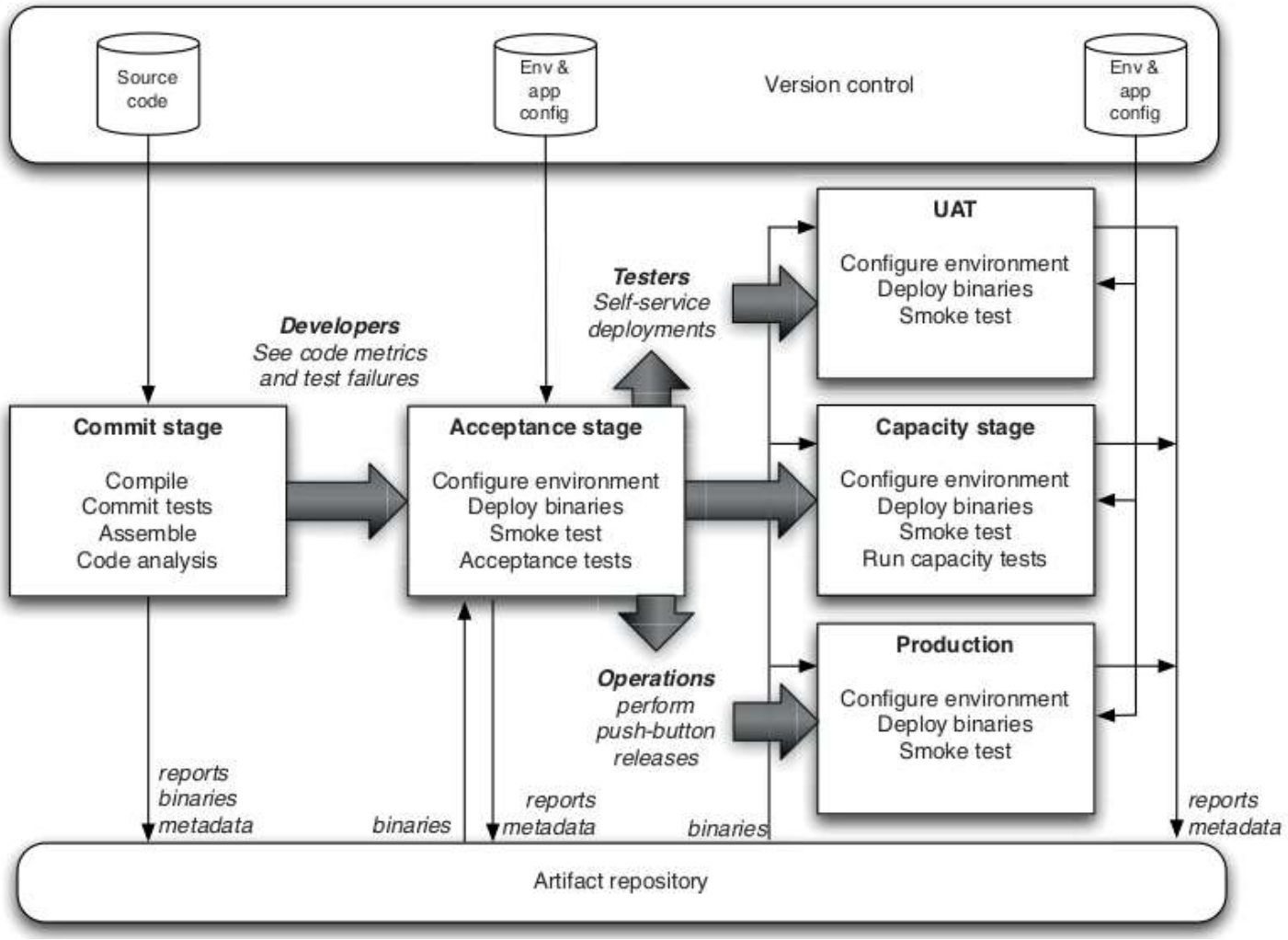
Deployment pipeline - Estructura

- Manual test
  - Verifica que el sistema es usable y cumple las necesidades del usuario
  - Puede detectar defectos no encontrados por el test automatizado
  - Verifica que provee valor al usuario
  - Test exploratorio, UAT

- Release stage
  - Entrega el sistema al usuario
  - Empaquetado o deployandolo directamente a producción o staging

Deployment pipeline - Estructura

- Deployment pipeline típico

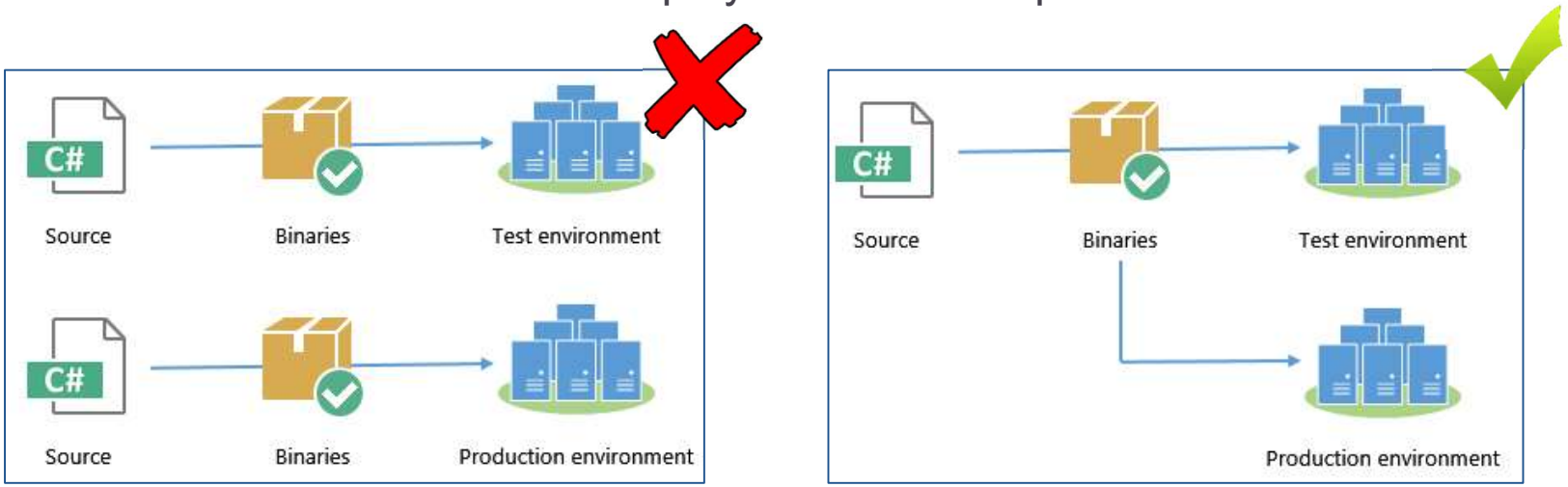


Ejemplo


Pipeline Activity - Cruise
https://demo.studios.thoughtworks.com/cruise/tab/pipeline/history/Demo
Pipeline Activity in Trader
Trader
Commit
Acceptance
Performance
UAT
Prod
1.2.86
revision: 86
10 minutes ago
by dfarley
1.2.85
revision: 85
1 hour ago
by jhumble
1.2.84
revision: 84
2 hours ago
by jhumble
Subversion - http://chistdcrsdmo01/svn/demo/trunk/
jhumble
#14 Fix performance problem
84
1.2.82
revision: 82
1 day ago
by dfarley
1.2.81
revision: 81
1 day ago
by jhumble
1.2.80
revision: 80
1 day ago
by jhumble
auto
manual
Done
demo.studios.thoughtworks.com

Deployment pipeline - Practices

- Solo construye los binarios una vez
  - Muchos sistemas consideran el VCS como la fuente canónica de cada paso, esto viola 2 principios
    - No es eficiente
    - No construye sobre lo conocido. Los binarios usados en prod deberían ser los mismos testeados en los demás environments
  - Re-creando binarios agrega riesgo innecesarios
  - Los binarios deben ser deployables en cualquier entorno



Deployment pipeline - Practices

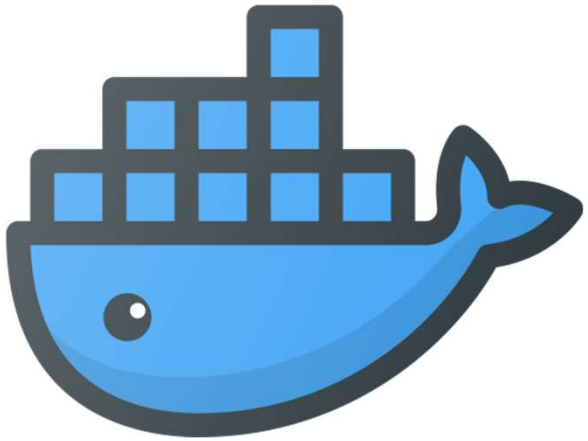
- Deploya de la misma manera en todo los entornos
  - Para testear también el proceso de deployment
  - Fuerza la separación entre código y configuración
  - Previene el sindrome works-in-my-machine

Deployment pipeline - Practices

- Smoke-test todos los deployments
  - Para verificar que tu aplicación está up & running

Deployment pipeline - Practices

- Deploya en una copia de producción
  - Para evitar problemas
  - Usar virtualización o containerización



Deployment pipeline

- ✧Cada cambio debería propagarse por el pipeline instanstaneamente
  - ✧Usar tools que permitan scheduling inteligente
- ✧Si cualquier parte del pipeline falla detén el pipeline
  - ✧Feedback rápido

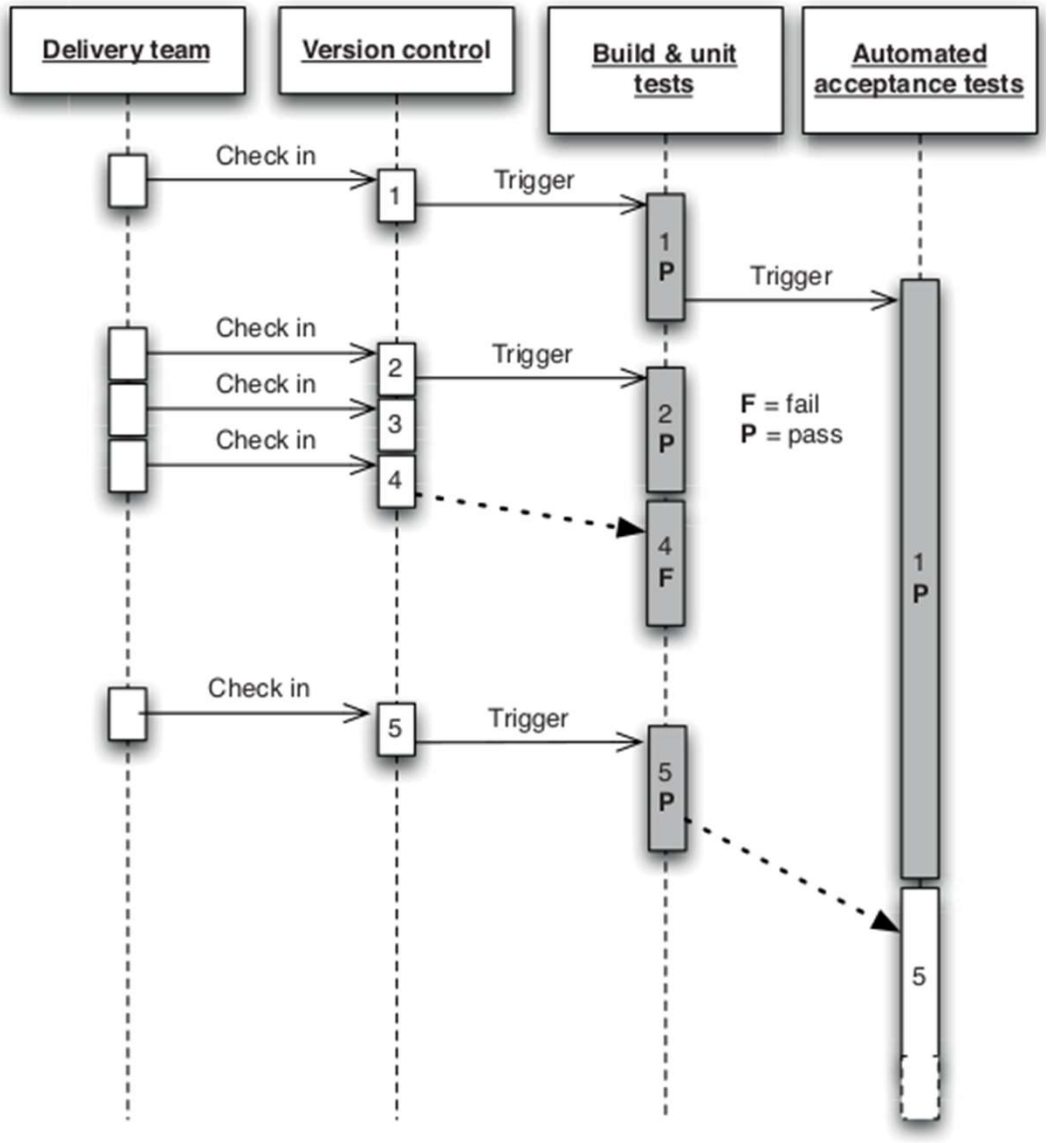
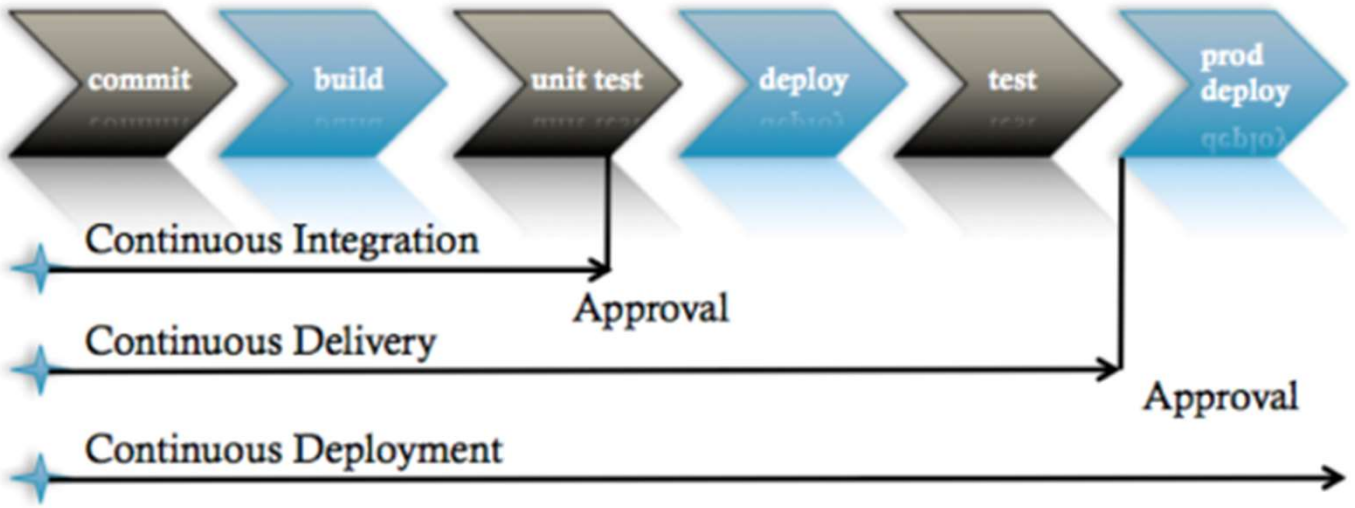


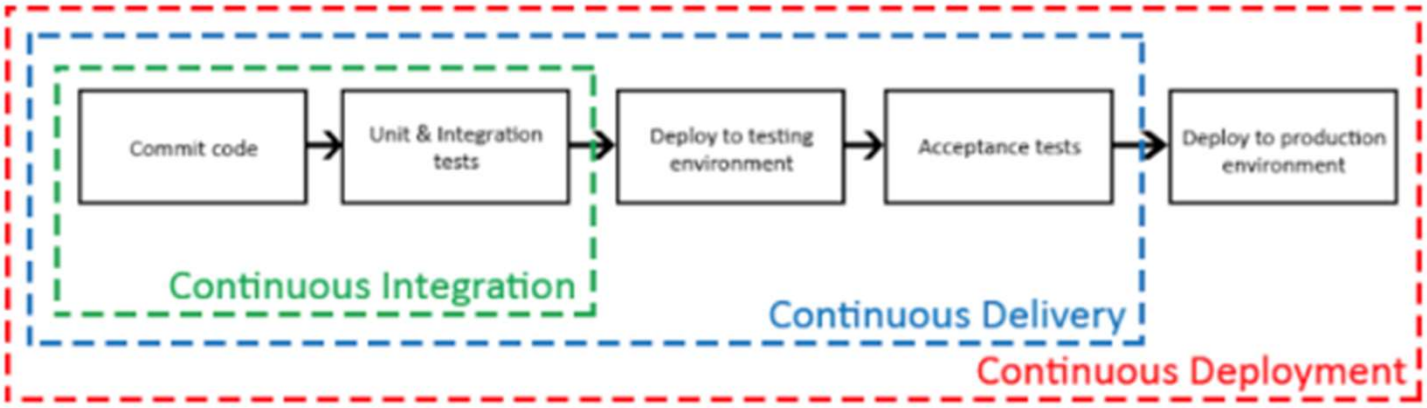
Figure 5.6 *Scheduling stages in a pipeline*

The Origin of the Term "Deployment Pipeline"

When we first used this idea, we named it a pipeline not because it was like a liquid flowing through a pipe; rather, for the hardcore geeks amongst us, it reminded us of the way processors "pipeline" their instruction execution in order to get a degree of parallelism. Processor chips can execute instructions in parallel. But how do you take a stream of machine instructions intended to be executed serially and divide them up into parallel streams that make sense? The way processors do this is very clever and quite complex, but in essence they often come to points where they effectively "guess" the result of an operation in a separate execution pipeline and start executing on the assumption of that guess. If the guess is later found to be wrong, the results of the stream that was based on it are simply dumped. There has been no gain—but no loss either. However, if the guess was good, the processor has just done twice as much work in the time it would take a single stream of execution—so for that spell, it was running twice as fast.

Our deployment pipeline concept works in the same way. We design our commit stage so that it will catch the majority of problems, while running very quickly. As a result, we make a "guess" that all of our subsequent test stages will pass, so we resume work on new features, preparing for the next commit and the initiation of the next release candidate. Meanwhile, our pipeline optimistically works on our assumption of success, in parallel to our development of new features.


commit
build
unit test
deploy
test
prod deploy
Continuous Integration
Approval
Continuous Delivery
Approval
Continuous Deployment

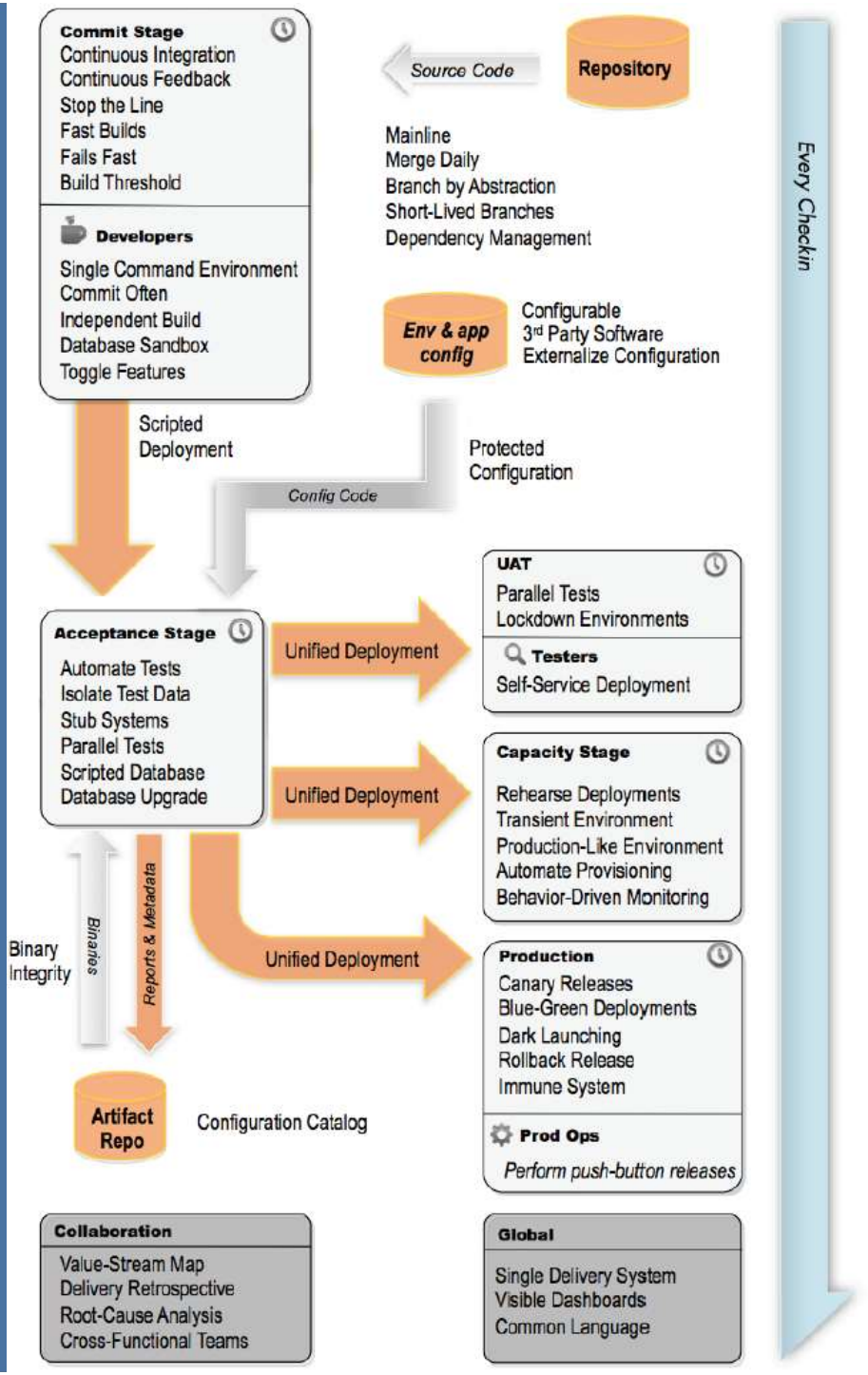

Commit code
Unit & Integration tests
Deploy to testing environment
Acceptance tests
Deploy to production environment
Continuous Integration
Continuous Delivery
Continuous Deployment

Deployment pipeline

- CI es un gran avance
  - Pero se enfoca en el desarrollo
  - Testing y Operations?
  - The last mile?

- La solución es adoptar un enfoque end-to-end

- ¿Qué es deployment pipeline?
  - Es una manifestación automatizada del proceso para llevar el sofware desde el control de versiones a las manos del usuario



Fin