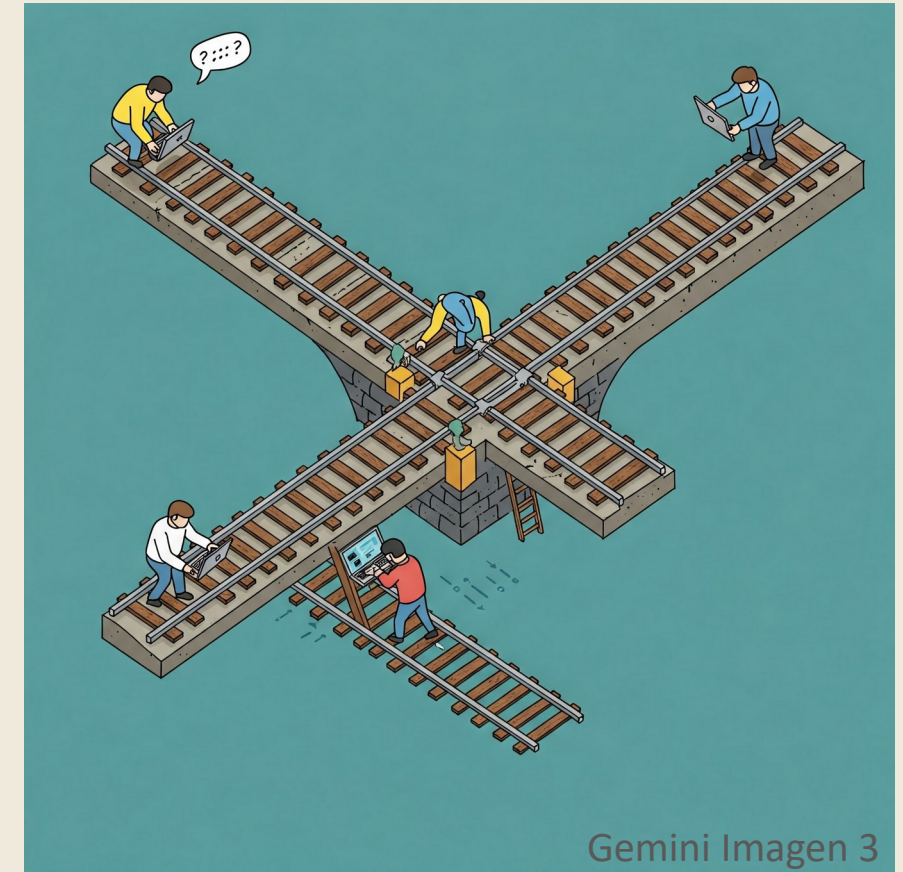
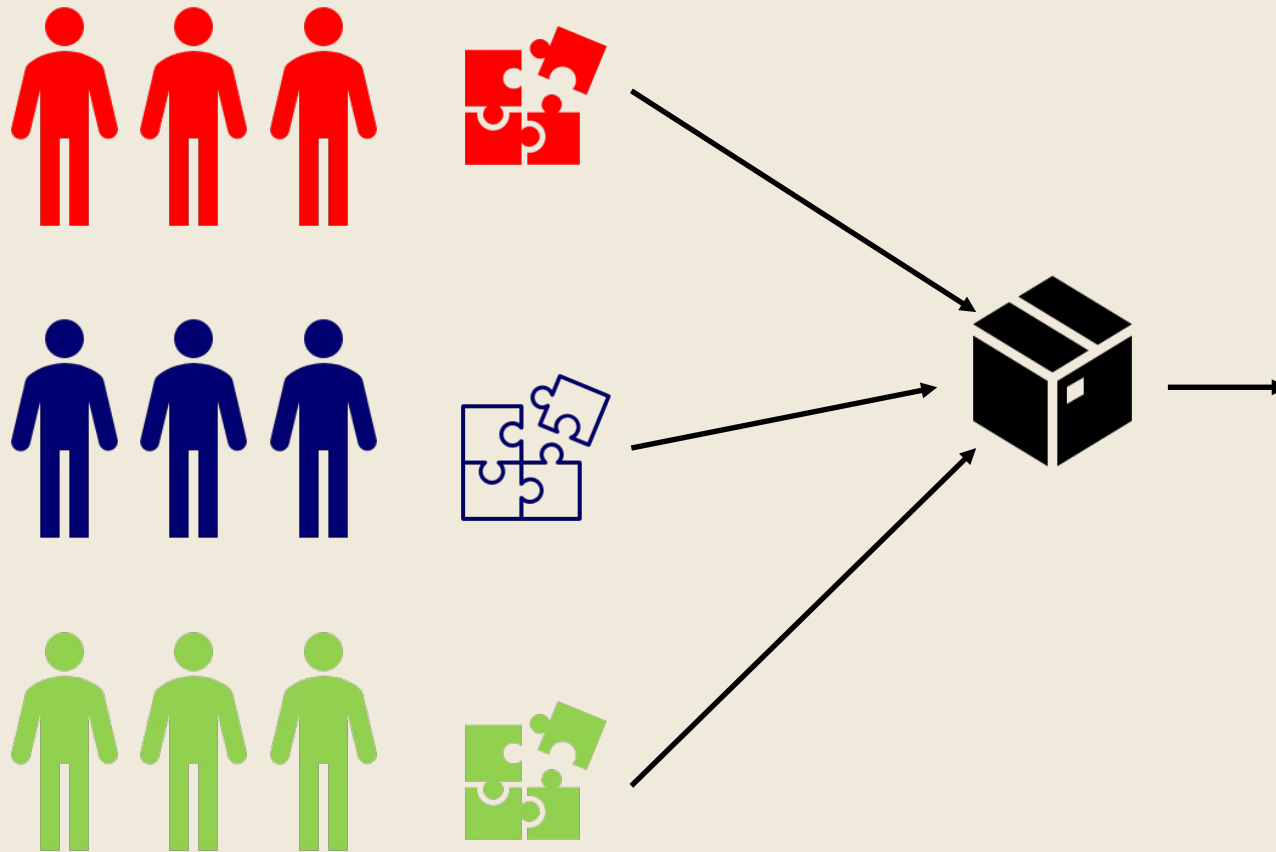


Unidad 2: Dominando la Integración Continua (CI)

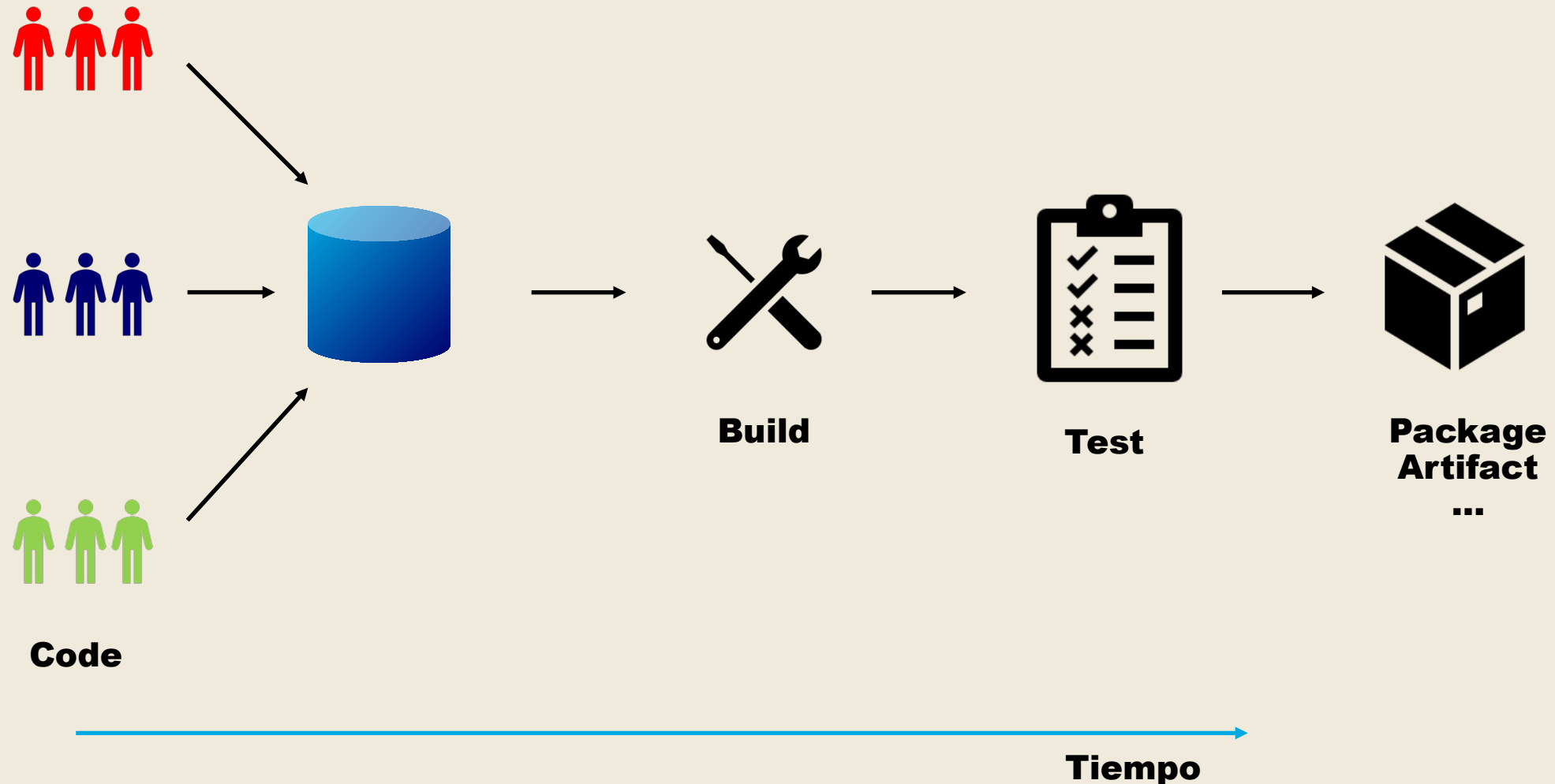
Retomando... ¿Qué pasa en DEV?



Gemini Imagen 3

Tiempo

¿Qué hace la integración continua?



¿Qué es Integración Continua (CI)?

Definición

Práctica donde los desarrolladores **integran** su **código** a un **repositorio compartido** frecuentemente (idealmente, varias veces al día).

Objetivo

Detectar errores de integración de forma temprana y rápida.

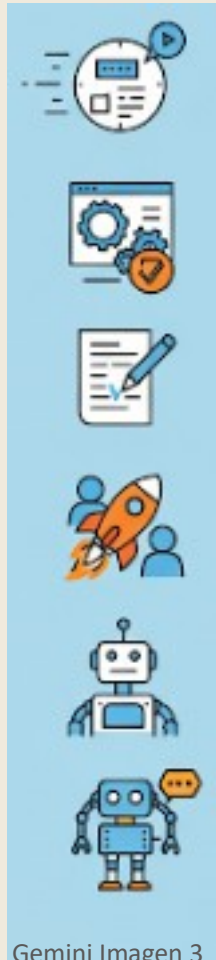
Verificación

Cada integración dispara una **build** automatizada y **pruebas automatizadas**.



¿Por qué CI?

Beneficios Clave



Gemini Imagen 3

→ Feedback Rápido























→ Menos Bugs

→ Mayor Velocidad

→ Facilidad de Integración

→ Automatización
(tiempo, error humano)


Principios Fundamentales de la integración continua

- ◆  **Control de Versiones Único:** Todo el  código fuente en un solo lugar (e.g.,  Git).
- ◆  **Automatizar la Build:**  Script que compila y  empaqueta de forma fiable.
- ◆  **Automatizar las Pruebas:** La  build se  auto-prueba para  validar corrección.
- ◆  **Integración Frecuente:**  Commits pequeños y  diarios al main o ramas de corta duración.
- ◆  **Build Rápida:** El proceso completo ( build +  tests rápidos) debe durar  minutos.
- ◆  **Feedback Visible:**  Resultados claros y accesibles para todo el  equipo.
- ◆  **Arreglar Builds Rotas Inmediatamente:** ¡ Prioridad máxima! ("🛑 Stop the line").



Automatización

El corazón de CI

¿Qué se automatiza?

- 
- Checkout de Código
 - Compilación
 - Ejecución de pruebas (unitarias, integración, análisis estático)
 - Empaquetado
 - Notificaciones

Beneficios Directos

- 
- **Consistencia:** Elimina el "funciona en mi máquina".
 - **Velocidad:** Elimina el "funciona en mi máquina".
 - **Fiabilidad:** Reduce errores humanos
- 

Herramientas de CI/CD


¿Qué nos proveen?

- Orquestan el pipeline automatizado.
- Detectan cambios en el repositorio, ejecutan los pasos definidos y reportan resultados.
- Algunas, proveen módulos de planeación y metodologías ágiles.

¿Cómo elegir?

- La elección depende de las necesidades, ecosistema (integración/compatibilidad) y presupuesto.

Ejemplos:

- **Jenkins** (Open Source, muy flexible)
 - **GitLab CI/CD** (Integrado con GitLab)
 - **GitHub Actions** (Integrado con GitHub)
 - **Azure DevOps**
 - **Google Cloud Build**
 - **CircleCI o Travis CI** (Populares en la nube)
- 

Pruebas automatizadas

“La red de seguridad”

Propósito:

- Validar que cada cambio no introduce regresiones ni rompe la funcionalidad existente.
- ¡Evita que el código defectuoso llegue a producción!



Características

- Son la **clave para tener confianza** en el proceso de integración continua y en el código.
- **CI sin pruebas automatizadas** robustas es solo una «integración frecuente» muy peligrosa.
- Un conjunto de **pruebas bien diseñadas** permite refactorizar y **añadir funcionalidades con seguridad**.



La pirámide de Pruebas

¿Cómo distribuir el esfuerzo de las pruebas automatizadas?

Pruebas End 2 End (UI)

Simulan flujo de usuario completo.
Lentas, frágiles, caras. Usar con moderación.

Pruebas de Integración / Aceptación

Verifican interacción entre componentes (e.g., servicio con BD). Más lentas, más complejas

Pruebas Unitarias

Rápidas, aisladas, baratas. Verifican componentes pequeños. Corren en cada commit.

Muchas pruebas rápidas en la base, menos pruebas lentas en la cima. Optimiza velocidad de feedback.



Pruebas Unitarias

Unit tests

Características:

- Verifican la unidad más pequeña de código (función, método, clase) de forma *aislada*.
- Usan **Mocks** o Stubs para **simular dependencias** externas.
- Son muy **rápidas, fáciles de escribir y mantener** (idealmente).
- **Cobertura de código:** Métrica que se genera para ver que código ha sido probado (no es infalible).

Rol en CI:

- Se ejecutan primero tras la compilación.
Feedback casi instantáneo al desarrollador.

Implementación (AAA):

Arrange (Define)

$A = 1$

$B = 2$

Act (Actua)

$C = \text{multiplicar } (A * B)$

Assert (Valida)

$C == 2$

Si la función a evaluar es demasiado sencilla puede no usar la estrategia AAA.

Pruebas de Integración / Aceptación

Pruebas de Integración:

- Verifican que diferentes unidades/módulos/servicios funcionan bien juntos.
- Ejemplos: Llamada a API real (en entorno de DEV), interacción con base de datos.
- Más lentas que las unitarias, pueden necesitar setup (e.g., BD en Docker).

Pruebas de Aceptación:

- Validan requisitos de negocio desde la perspectiva del usuario/cliente.
- A menudo implementadas como pruebas E2E o usando BDD (Behavior-Driven Development).
- Herramientas BDD: Karate, Selenium, Cucumber, SpecFlow (ayudan a definir pruebas en lenguaje natural).



- En el taller de CI vamos a realizar pruebas de aceptación combinando Selenium (para pruebas de aplicaciones web) y pytest –
 - Las pruebas de integración no las cubriremos de manera práctica pues implican un entorno más complejo –
 - Las pruebas de aceptación, **cuando se tiene un ciclo de CI/CD completo, pasan a ser parte de CD** –

Calidad y Seguridad del Código

Análisis Estático

Definición:

Análisis del código fuente sin ejecutarlo.
Se integra temprano en el pipeline.

Beneficio:

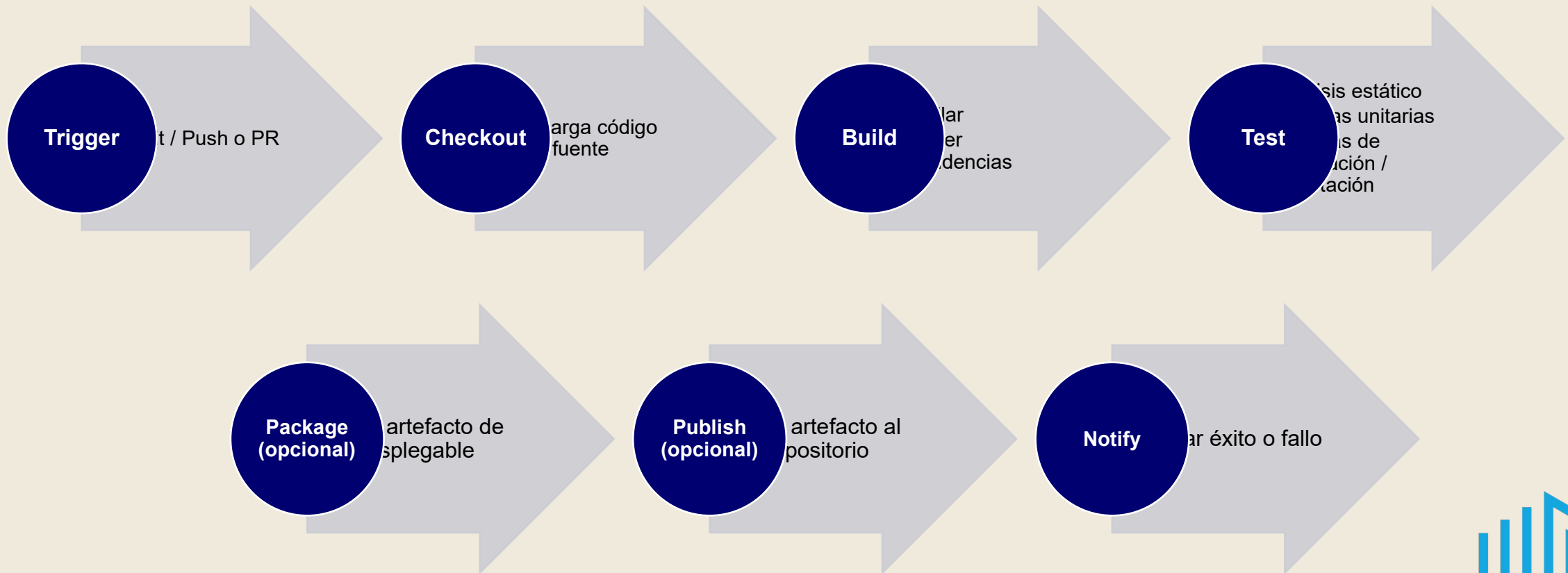
Feedback inmediato sobre calidad y seguridad antes de fases más lentas.

Tipos:

- **Linters/Formateadores:** Estilo de código consistente (ESLint, Prettier, Black).
- **Análisis de Complejidad:** Detecta código difícil de entender/mantener.
- **Detección de Bugs Potenciales:** Patrones de código problemáticos.
- **SAST (Static Application Security Testing):** Busca vulnerabilidades de seguridad conocidas en el código (e.g., inyección SQL, XSS).
- **SCA (Software Composition Analysis):** Analiza dependencias de terceros en busca de vulnerabilidades conocidas.

El pipeline de CI: Flujo típico

(vista general)



Fase 1: Code

Control de versiones y ramas



Git → El estándar de facto para versionar código fuente.

Commits pequeños y frecuentes → Facilitan la integración y depuración.

Estrategias de ramificación (Branch)

- **Trunk-Based Development (TBD):** Recomendado para CI/CD. Desarrollo cerca de *main/trunk*, usando **ramas de corta duración** (<1 día) llamadas *Features*. Permite integración continua real.
- **GitHub Flow / Git Flow:** Es un modelo de ramificación que define un flujo estricto para *lanzamientos*, *hotfixes* y *desarrollo de funciones*, utilizando *múltiples ramas de larga duración*.

TBD prioriza la integración constante en una rama principal, mientras que **Git Flow** organiza el desarrollo en torno a distintas ramas para manejar lanzamientos y funcionalidades separadas.

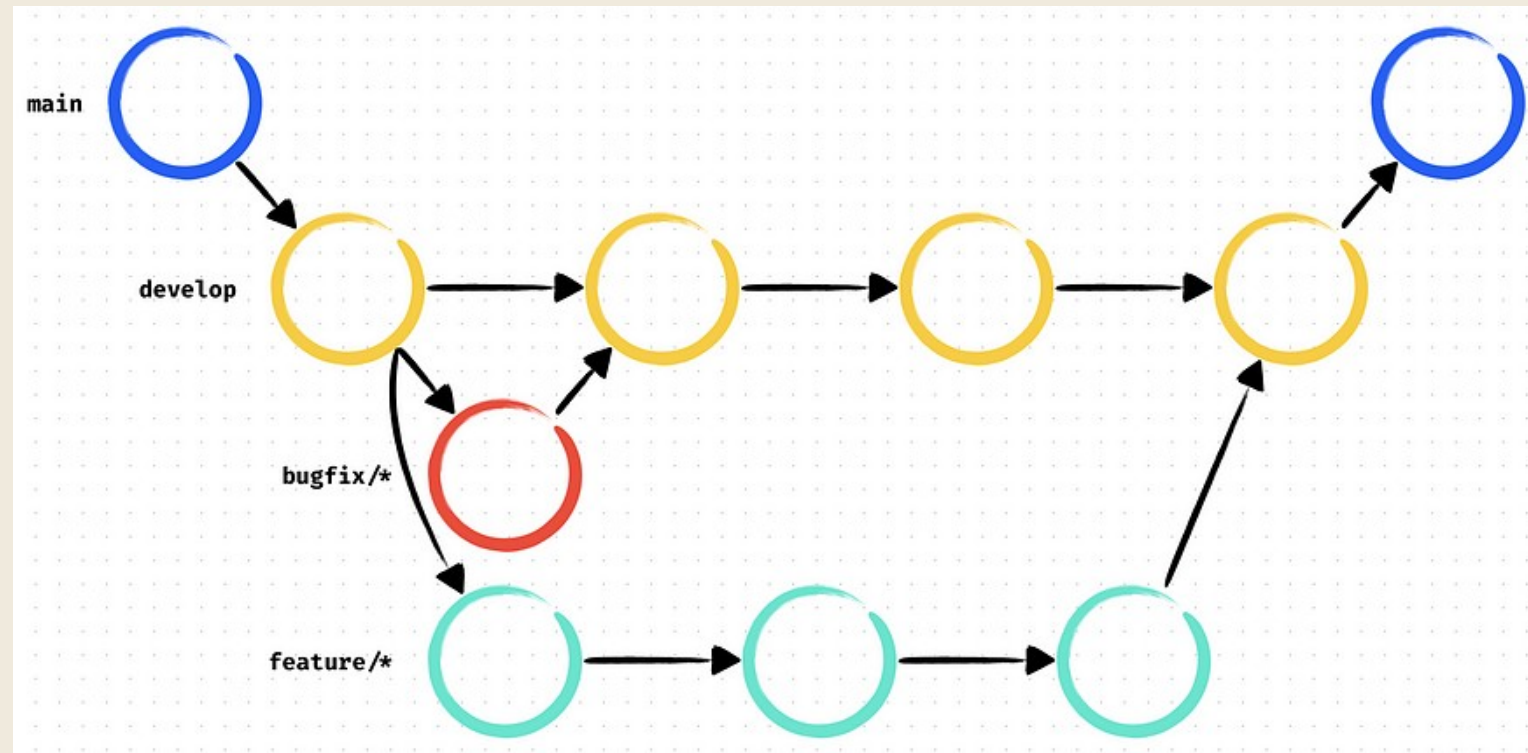
Revisión de código (vía Pull Request / Merge Request)

- Paso crucial antes del merge en rama main, trunk o principal para garantizar calidad y conocimiento compartido.

Fase 1: Code

Control de versiones y ramas

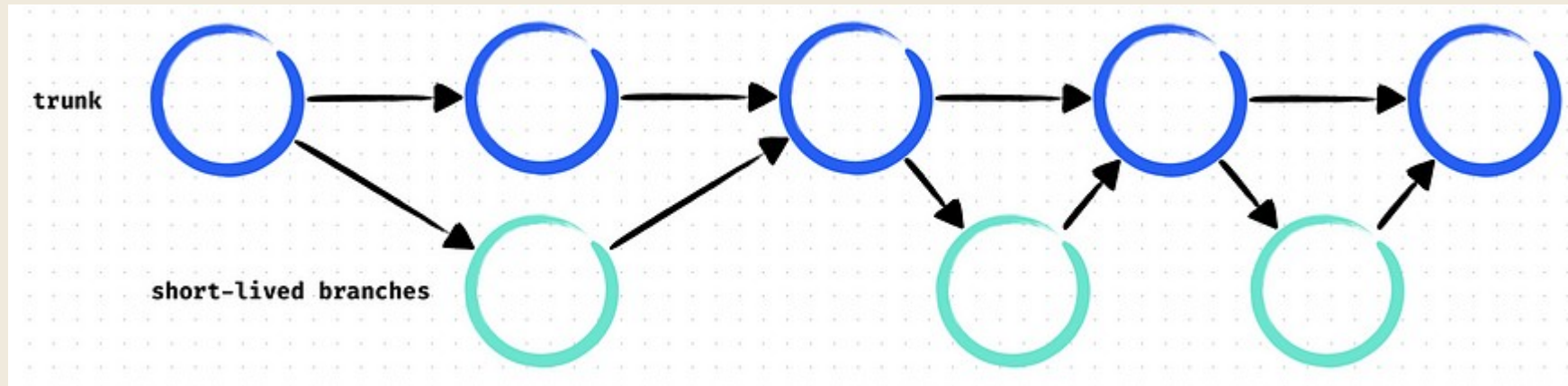
GIT FLOW



Fase 1: Code

Control de versiones y ramas

TRUNK BASED





Fase 2: Build

Compilación y Dependencias



OBJETIVO → Transformar código fuente en algo ejecutable o desplegable.

TAREAS → Compilar (Java -> .class / .jar) – Transpilar (TS -> JS) – Empaquetar (Python Package)
Implicítamente está la resolución de dependencias

GESTIÓN DE DEPENDENCIAS

→ **Herramientas:** Maven, Gradle, NPM, pip, NuGet, Poetry

→ **Lock Files:** Son archivos que registran las versiones exactas de las dependencias instaladas para garantizar reproducibilidad y consistencia (**muy importantes**).

DOCKER → Excelente para **definir y aislar el entorno** de build y **compilar en el mismo sistema** operativo y entorno de despliegue (si aplica).

**UN ENTORNO DE BUILD CONSISTENTE ES LA CLAVE
PARA EVITAR EL “EN MI MÁQUINA FUNCIONA”**

Fase 3: Test

Ejecución en el pipeline

OBJETIVO → Ejecutar las pruebas configuradas automáticamente.

ORDEN TÍPICO →

1. Análisis estático (calidad y seguridad – SAST / SCA)
2. Pruebas unitarias (+ cobertura)
3. Prueba de integración / aceptación



QUALITY GATES →

Criterios automáticos que deben pasar para que el pipeline continúe. Por ejemplo:

- Cobertura mínima del 90%
- 0 bugs críticos o inferiores a calificación “C”
- 100% de tests pasando

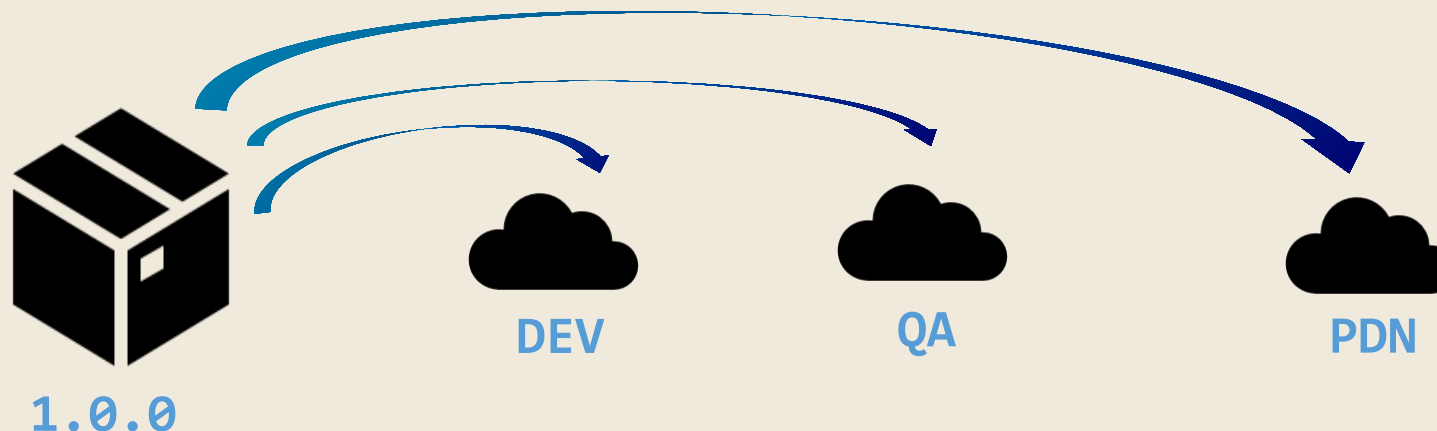
Herramientas como **SonarQube / SonarCloud** facilitan esto *¡la usaremos en el taller!*

SI ALGUNA ETAPA FALLA, EL PIPELINE SE DETIENE EN EL QUALITY GATE

Fase 4: Release

Creación de artefactos

- OBJETIVO** → Si Build y Test son exitosos, esta fase busca crear un artefacto: la unidad versionada, empaquetada y lista para ser desplegada.
- EJEMPLOS** → JAR, WAR, .WHL, Paquete NPM, Imagen Docker, Binario ejecutable, .ZIP
- INMUTABILIDAD** → El artefacto se construye **UNA VEZ** y luego se promueve a través de diferentes entornos (Testing, Staging, Producción). **¡No se reconstruye para cada entorno!**
- VERSIONAMIENTO** → Es crucial versionar los artefactos utilizando valores de versión: **SemVer, Commit Hash, Build Number**. Permite trazabilidad y rollback.





Gestión de artefactos

Repositorios

NECESIDAD → Los artefactos generados deben almacenarse en un **Repositorio de Artefactos**.

PROPÓSITO →

- **Almacenamiento** centralizado y versionado.
- **Compartir artefactos** entre pipelines o equipos.
- **Cachear dependencias** externas (acelera builds, mejora fiabilidad).
- **Gestionar seguridad y permisos**.

EJEMPLOS → JFrog Artifactory, Sonatype Nexus, Docker Hub/Registry, GitHub/GitLab Packages, AWS ECR, Google Artifact Registry.



JFrog Artifactory



Gestión de secretos en CI

¿QUÉ ES UN SECRETO? →

Información sensible necesaria para el build, test o despliegue (**API Keys, contraseñas de BD, tokens, certificados**).

¡PELIGRO! →



NUNCA almacenar secretos directamente en el código fuente (Git) ni en los scripts de configuración del pipeline en texto plano.

Riesgo → Exposición accidental puede llevar a brechas de seguridad graves.



Estrategias Seguras para Secretos

Variables de Entorno Seguras (del Servidor CI/CD)→

La mayoría de herramientas CI/CD permiten definir secretos que se inyectan como variables de entorno, ocultándolos en los logs. GitHub Actions es una de ellas.

Gestores de Secretos Dedicados (Recomendado)→

- **Herramientas:** HashiCorp Vault, AWS Secrets Manager, AzureKey Vault, Google Secret Manager.
- El pipeline se autentica con el gestor y obtiene los secretos necesarios dinámicamente en tiempo de ejecución.

Secretos Encriptados en Git (Avanzado/Precaución) →

- **Por ejemplo:** git-crypt, Mozilla SOPS.
- Requiere gestión cuidadosa de claves.



Notificaciones y visibilidad

SON CRUCIALES PARA QUE EL EQUIPO SEPA EL ESTADO DEL PIPELINE

Notificaciones →

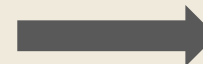
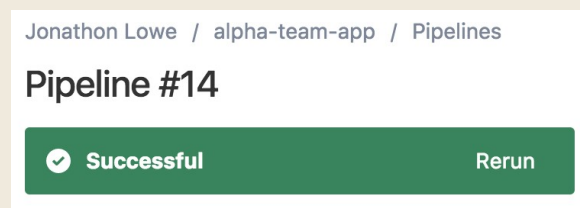
- Configurar alertas (Email, Slack, Teams) para builds exitosas y (especialmente) fallidas.
- Algunos servidores de CI/CD tienen notificaciones nativas, por correo por ejemplo.

Dashboards →

- Los servidores CI/CD ofrecen vistas del historial de builds, logs, resultados de tests.

Badges de Estado →

- Indicadores visuales (e.g., en README.md) del estado de la build de la rama principal.



¿Qué es la deuda técnica?

REFLEXIÓN: Tomar atajos hoy en **calidad**, te puede cobrar **intereses** en el futuro.

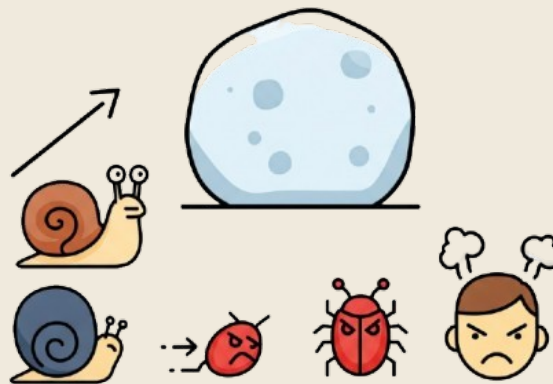
Definición →

- Es el resultado implícito de **priorizar la velocidad a corto plazo sobre la sostenibilidad** a largo plazo.
- Puede ser **deliberada** (decisión consciente) o **accidental** (malas prácticas).
- No es solo código malo, también **falta de automatización, documentación pobre, etc.**




Impacto de la Deuda Técnica

- **Ralentiza el Desarrollo:** Añadir features o arreglar bugs se vuelve más lento y costoso.
- **Aumenta los Defectos:** Código complejo y frágil es propenso a errores.
- **Dificulta la Contratación/Retención:** Frustra a los desarrolladores.
- **Incrementa el Riesgo:** Dificulta la adopción de parches de seguridad o nuevas tecnologías.
- **Impacta Métricas Clave:** Aumenta Lead Time, reduce Deployment Frequency, incrementa Change Fail Rate.





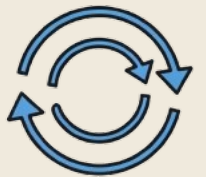
Integración continua y la gestión de la deuda técnica

- **CI Expone la Deuda:** Builds lentas, tests "flaky" (intermitentes), fallos frecuentes en el pipeline *son síntomas de deuda técnica*.
 - **CI Ayuda a Prevenir Nueva Deuda:**
 - Análisis estático detecta código complejo/inseguro temprano.
 - Pruebas unitarias fomentan diseño modular y testable.
 - Feedback rápido desincentiva dejar problemas "para después".
 - **CI Facilita Pagar la Deuda:**
 - La suite de tests da confianza para refactorizar código existente.
 - Permite introducir mejoras de forma incremental y validarlas rápidamente.
- 

Estrategias para Manejar la Deuda Técnica



- **Visibilidad:** Medirla (SonarQube), incluirla en el backlog. ¡Hacerla explícita!
- **Priorización:** Enfocarse en la deuda que más duele (más ralentiza, más riesgo genera).
- **Tiempo Dedicado:** Asignar capacidad del equipo regularmente (e.g., 10-20% del sprint) para refactorizar y mejorar. *"Regla del Boy Scout"*.
- **Refactorización Continua:** Pequeñas mejoras constantes son mejor que grandes reescrituras arriesgadas.
- **Mejorar Prácticas:** Fomentar TDD (Desarrollo Guiado por Pruebas), BDD (Desarrollo Guiado por Comportamiento), Code Reviews, Pair Programming.

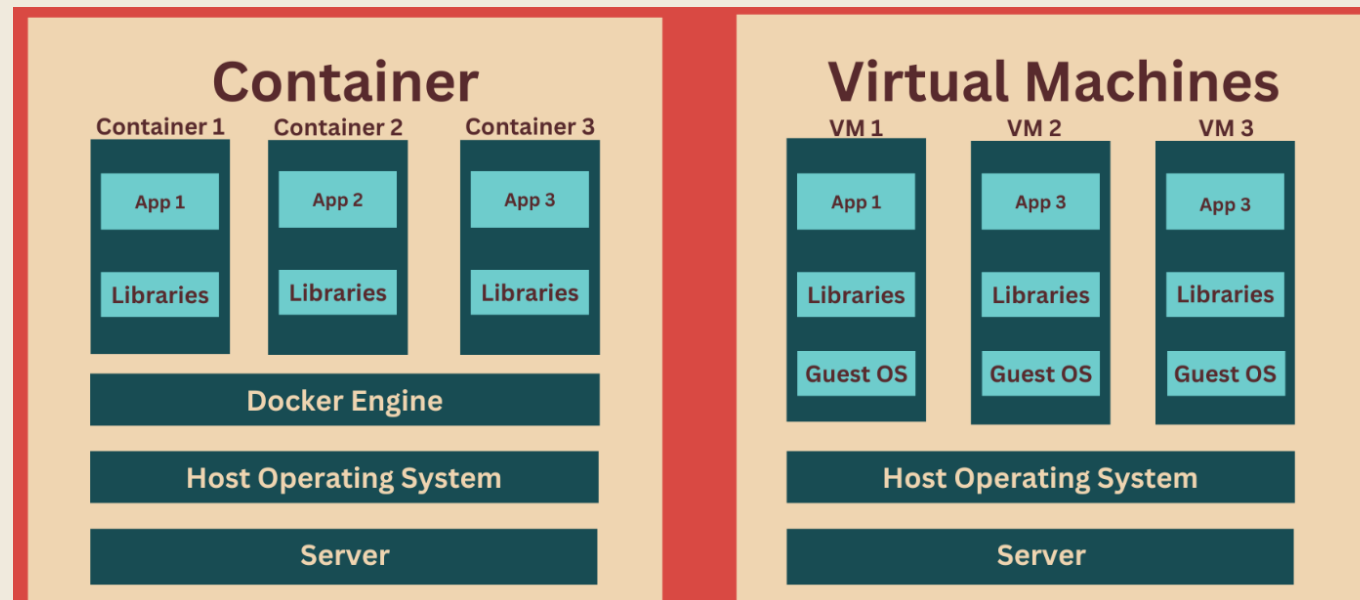


Concepto Calve

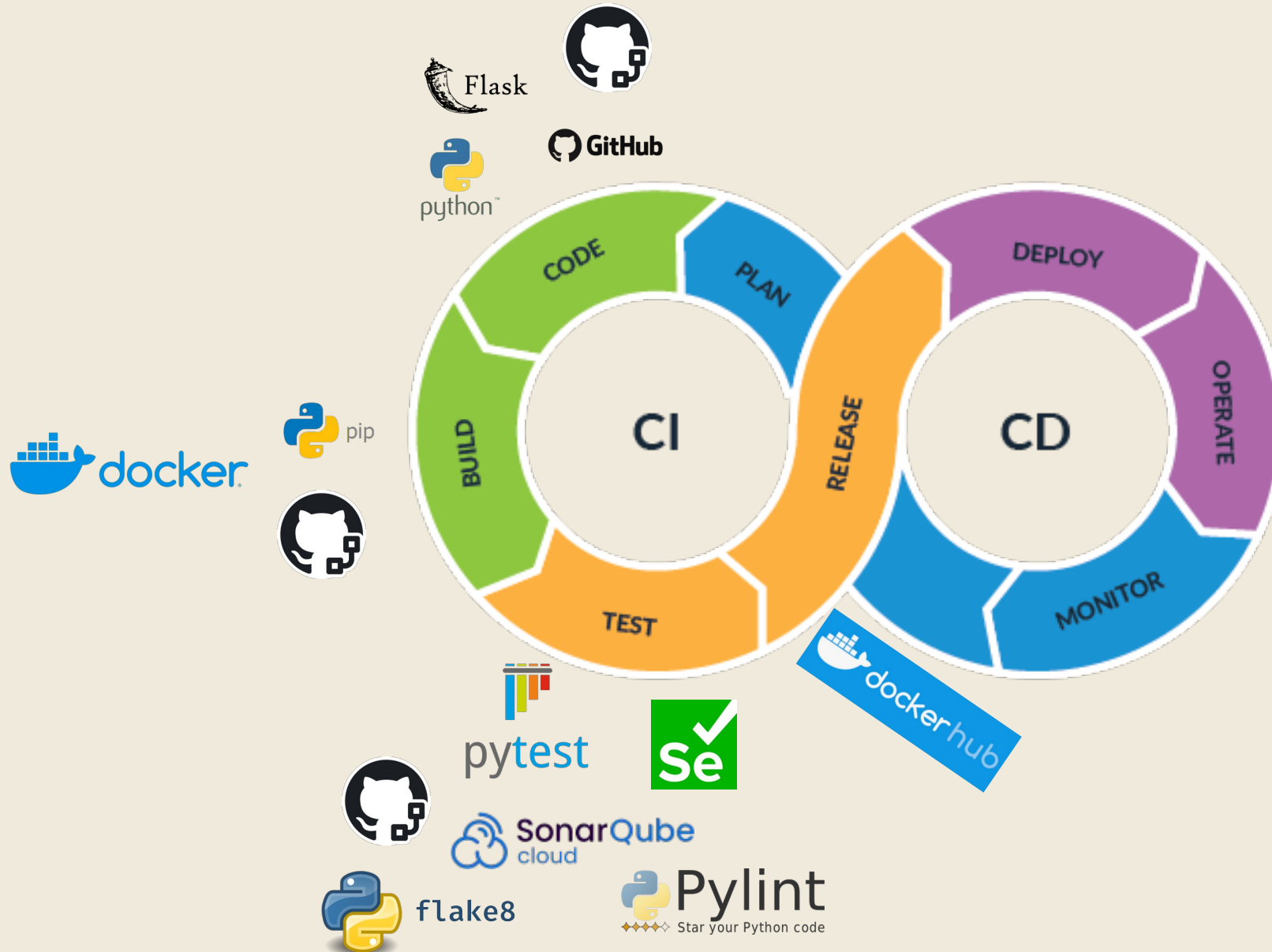
Previo al taller

Contenedores (Docker):

La contenerización **consiste en empaquetar aplicaciones y sus dependencias en contenedores aislados y portátiles**, lo que facilita su **ejecución consistente** en cualquier entorno, mejorando la eficiencia y la agilidad en el desarrollo y despliegue de software.

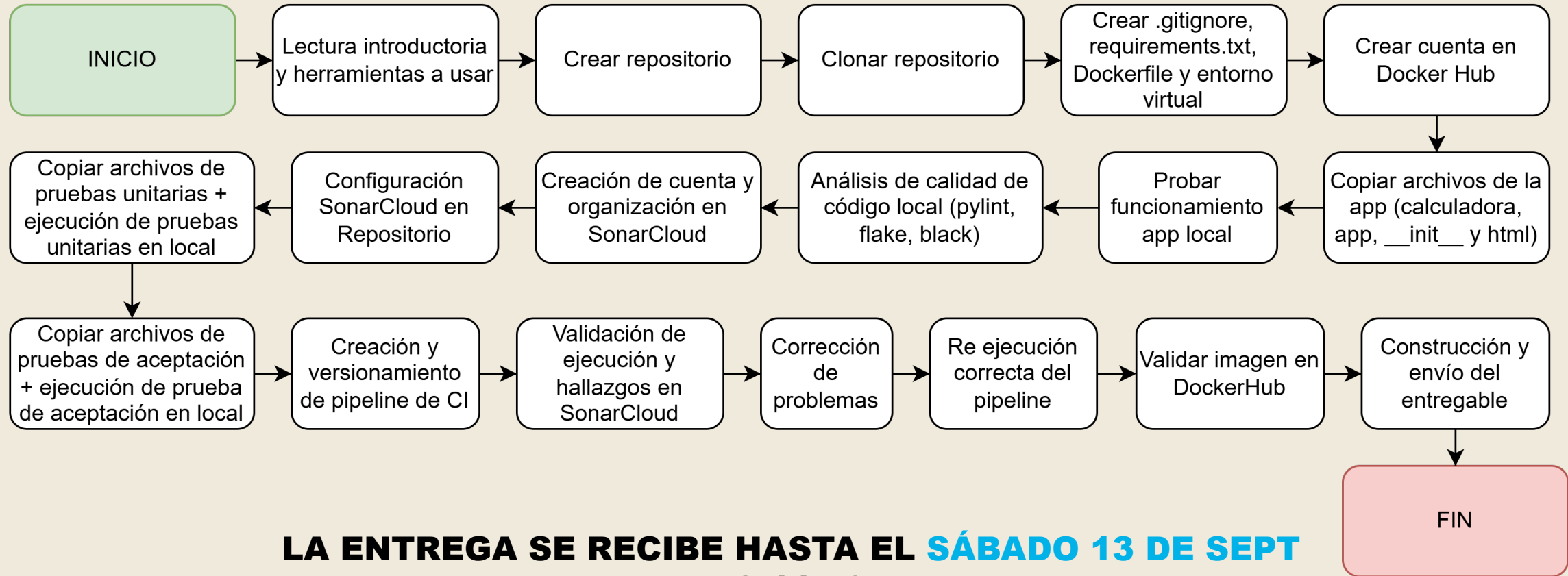


Herramientas del taller de CI



TALLER GRUPAL ENTREGABLE 2

Pipeline de CI con Python, GitHub Actions y Herramientas Open Source



**LA ENTREGA SE RECIBE HASTA EL SÁBADO 13 DE SEPT
A LAS 11:59 PM**

TALLER GRUPAL ENTREGABLE 2

Pipeline de CI con
Python, GitHub Actions y Herramientas Open Source

¿DUDAS?

¿PREGUNTAS?

¿INCONVENIENTES?



Resumen Clave del Módulo

- CI = Integrar frecuentemente + Validar automáticamente (Build + Test).
- Automatización es la base para velocidad, consistencia y fiabilidad.
- Pruebas (Unit, Integ, Static) son la red de seguridad esencial.
- El pipeline CI orquesta el flujo: Code -> Build -> Test -> (Package/Publish) -> Notify
- Gestionar artefactos (repos) y secretos (vaults) es crucial.
- CI ayuda a exponer, prevenir y gestionar la deuda técnica.



Preguntas y Discusión


¿Cuál creen que es el mayor desafío al implementar CI por primera vez en un equipo/organización?

¿Respecto a la deuda técnica, que herramienta del taller crees que indica y acumula la deuda?

¿Si necesitas que un pipeline de CI pase, y conscientemente aceptas los riesgos, que deberías hacer en el pipeline del taller para permitir su ejecución?

¿Qué opinas de los hallazgos de seguridad reportados?

¿Algo en particular que resaltar o consultar de este módulo de CI?



Referencias Clave

Fundamentales →

- "Continuous Delivery" – Humble, Farley
- "Accelerate" – Forsgren, Humble, Kim
- "The DevOps Handbook" – Kim, Debois, Willis, Humble
- "The Phoenix Project" / "The Unicorn Project" – Gene Kim et al.


Técnicos / Implementación →

- "Infrastructure as Code" – Morris
- "Docker Deep Dive" – Poulton
- Documentación oficial: Docker, Kubernetes, Terraform, Servidores CI/CD.

Seguridad →

- "DevSecOps" - Wilson

Observabilidad →

- "Site Reliability Engineering" – Google/O'Reilly
 - "Observability Engineering" - Majors, Fong-Jones, Miranda
- 

**Inspira
Crea
Transforma**

www.eafit.edu.co

VIGILADA | MINEDUCACIÓN

**UNIVERSIDAD
EAFIT**