

Integración Continua

Ingeniería de Software 2 (CC3091) - 2026

Integración Continua

Semestre 02, 2026

Introducción

La integración de código es la actividad más riesgosa de un proyecto de software con varios desarrolladores, porque es el momento en que el trabajo de todos se junta por primera vez.

La integración continua parte de una idea simple. Si una actividad resulta costosa y dolorosa, hay que hacerla más seguido hasta volverla trivial.

No es una herramienta ni un servidor, es una práctica de equipo que se sostiene con herramientas.

Integración Continua

Martin Fowler la define como la práctica en la que cada miembro del equipo integra su trabajo a un repositorio compartido con frecuencia, al menos una vez al día, y cada integración se verifica con una compilación y una batería de pruebas automáticas.

La definición tiene tres partes inseparables. Integrar seguido al main branch, verificar cada integración de forma automática y corregir de inmediato lo que se rompa.

Un equipo que tiene un servidor de CI pero integra cada dos semanas no practica integración continua, solo tiene un servidor de CI.

El infierno de la integración

Cuando cada desarrollador trabaja aislado durante semanas, el código divergente crece sin que nadie lo note.

- **Conflictos semánticos:** Git puede fusionar dos archivos sin conflicto de texto y aun así producir un programa que no funciona, porque las dos ramas cambiaron supuestos distintos sobre la misma función.
- **Costo no lineal:** el esfuerzo de fusionar no crece con el tamaño del cambio, crece más rápido que él, así que una rama de un mes cuesta mucho más que cuatro ramas de una semana.
- **Fase de estabilización:** los proyectos que integran tarde necesitan una etapa final impredecible dedicada solo a hacer que el sistema vuelva a funcionar.
- **Miedo a refactorizar:** si integrar duele, nadie se atreve a mejorar la estructura del código, y la deuda técnica se acumula.

Desarrollo sobre el main branch

La integración continua exige que exista una sola rama principal, llamada main branch, que se mantiene siempre funcional.

El trabajo se hace en ramas de vida corta, de horas o de un día, que se fusionan al main branch el mismo día en que se abren.

Las ramas de larga vida son el antipatrón clásico. Mientras más tiempo vive una rama, más divergente está del main branch y más caro resulta el merge.

Cuando una funcionalidad es demasiado grande para terminarse en un día, se integra incompleta pero inactiva, con un interruptor de funcionalidad (feature flag) que la mantiene apagada en producción.

Prácticas de la integración continua

Fowler enumera un conjunto de prácticas que en conjunto hacen posible la integración continua.

- **Un solo repositorio:** todo lo necesario para construir el sistema vive en control de versiones, incluido el script de compilación y la configuración.
- **Compilación automatizada:** un solo comando compila, configura y produce el sistema ejecutable, sin pasos manuales ni conocimiento tribal.
- **Build auto-verificable:** la compilación incluye una batería de pruebas automáticas, y si alguna falla la compilación falla.
- **Integrar al main branch todos los días:** cada desarrollador empuja su trabajo al main branch al menos una vez al día.
- **Verificar cada commit en un entorno de referencia:** el servidor de CI construye cada commit del main branch en una máquina limpia, no en la laptop del desarrollador.
- **Corregir el build roto de inmediato:** un main branch roto bloquea a todo el equipo, así que arreglarlo tiene prioridad sobre cualquier otra tarea.

- **Mantener el build rápido:** la meta es diez minutos o menos para la compilación de commit, porque un pipeline lento deja de usarse.
- **Probar en una copia de producción:** mientras más parecido sea el entorno de pruebas al de producción, menos defectos se escapan.
- **Estado visible para todos:** cualquiera debe poder saber en segundos si el main branch está en verde y quién hizo el último cambio.
- **Despliegue automatizado:** el mismo artefacto verificado se promueve entre entornos con un script, no a mano.

El build auto-verificable

La pieza que distingue a la integración continua de una simple compilación programada es la batería de pruebas.

Sin pruebas automáticas, el servidor de CI solo confirma que el código compila, y compilar no significa funcionar.

La batería se organiza en capas. Las pruebas unitarias son muchas, rápidas y aisladas; las de integración verifican la conversación entre componentes; las de extremo a extremo son pocas, lentas y frágiles.

La pirámide de pruebas es una guía de proporción. Mayoría de pruebas unitarias en la base, menos pruebas de integración en el medio y muy pocas pruebas de interfaz en la punta.

Anatomía de un pipeline de CI

Un pipeline es la secuencia de etapas que se ejecuta de forma automática después de cada commit.

- **Disparador:** el evento que inicia el pipeline, normalmente un push al main branch o la apertura de un pull request.
- **Preparación del entorno:** se descarga el código en una máquina limpia y se instalan las dependencias con versiones fijas.
- **Compilación:** se produce el artefacto ejecutable una sola vez, y ese mismo artefacto es el que viaja por el resto del pipeline.
- **Pruebas:** se ejecuta la batería automática, empezando por las rápidas para fallar temprano.
- **Análisis estático:** linters, formateadores y analizadores de calidad revisan el código sin ejecutarlo.
- **Análisis de dependencias:** se revisan las bibliotecas de terceros en busca de vulnerabilidades conocidas.
- **Publicación del artefacto:** el paquete verificado se guarda en un repositorio de artefactos con un número de build.
- **Notificación:** el resultado se informa al equipo por el canal que ya usa, preferiblemente chat en vez de correo.

Servidores de integración continua

Todas estas herramientas cumplen la misma función, ejecutar un pipeline definido en un archivo versionado dentro del repositorio.

- **GitHub Actions:** integrado a GitHub, se configura con archivos YAML en `.github/workflows` y cuenta con un mercado de acciones reutilizables. Es la opción por defecto cuando el código ya está en GitHub.
- **GitLab CI/CD:** parte de la plataforma GitLab, se configura en `.gitlab-ci.yml` y ofrece la experiencia más completa en un solo producto, con repositorio, pipelines y escaneo de seguridad juntos. Disponible en la nube y autoalojado.

- **Jenkins:** servidor de código abierto autoalojado, con un ecosistema enorme de plugins y pipelines definidos en `Jenkinsfile` . Es el más flexible y también el que más mantenimiento exige.
- **CircleCI:** servicio en la nube orientado a la velocidad, con paralelismo, caché y entornos Docker, VM y ARM. Se elige cuando el tiempo de build es una restricción medible.
- **Azure Pipelines:** parte de Azure DevOps, con buena integración al ecosistema Microsoft y agentes para Windows, Linux y macOS.
- **Bitbucket Pipelines:** opción nativa para equipos que ya usan Bitbucket y Jira, configurada en `bitbucket-pipelines.yml` .

Herramientas de apoyo del pipeline

El servidor de CI solo orquesta. El trabajo real lo hacen las herramientas que se invocan en cada etapa.

- **Compilación y dependencias:** Maven y Gradle en Java, npm y pnpm en JavaScript, Composer en PHP, pip y uv en Python, Cargo en Rust.
- **Pruebas:** JUnit en Java, Pest y PHPUnit en PHP, pytest en Python, Jest y Vitest en JavaScript, Playwright y Cypress para extremo a extremo.
- **Cobertura:** JaCoCo, coverage.py y Codecov miden qué porcentaje del código ejecutan las pruebas.
- **Análisis estático y formato:** SonarQube, ESLint, Prettier, Black y Checkstyle revisan calidad y estilo de forma automática.
- **Seguridad:** Dependabot y Renovate actualizan dependencias, OWASP Dependency-Check y Trivy buscan vulnerabilidades conocidas, Gitleaks detecta secretos filtrados en el repositorio.

- **Contenedores y artefactos:** Docker empaqueta el sistema, y registros como GitHub Container Registry o Artifactory almacenan cada build con su versión.

Un workflow de GitHub Actions

```
name: CI
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
jobs:
  verificar:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '22'
      - run: npm ci
      - run: npm run lint
      - run: npm test
```

- La clave `on` define los disparadores. Este pipeline corre en cada push al main branch y en cada pull request dirigido al main branch, que son los dos momentos donde importa saber si el sistema sigue funcionando.
- La clave `jobs` agrupa las unidades de trabajo, y `runs-on` declara la máquina limpia donde se ejecutan, provista por GitHub en cada corrida.
- Los pasos con `uses` invocan acciones reutilizables. `checkout` descarga el código del commit que disparó el pipeline y `setup-node` instala la versión exacta del lenguaje.

- Los pasos con `run` ejecutan comandos. `npm ci` instala dependencias con versiones fijas, `npm run lint` corre el análisis estático y `npm test` ejecuta la batería de pruebas.
- Si cualquier paso termina con código distinto de cero, el job falla completo y GitHub marca el commit en rojo.

Integración en el ciclo de vida de desarrollo

La integración continua no es una etapa del ciclo de vida, es una práctica que atraviesa todas sus etapas.

- **Requerimientos:** los criterios de aceptación de cada historia se escriben de forma verificable, para poder convertirse en pruebas automáticas.
- **Diseño:** la arquitectura debe permitir probar los componentes por separado, con dependencias inyectables y módulos desacoplados.
- **Construcción:** el desarrollador integra al main branch a diario, y el pipeline le devuelve retroalimentación en minutos en vez de semanas.
- **Verificación y validación:** la mayor parte de las pruebas de regresión dejan de ser manuales y pasan a ejecutarse en cada commit.
- **Despliegue:** el artefacto que el pipeline verificó es el mismo que se promueve a producción, sin recompilar.
- **Operación y mantenimiento:** cada defecto reportado en producción se documenta con una prueba automática que falla, y esa prueba entra al pipeline para que el defecto no vuelva.

El flujo de trabajo del desarrollador

La práctica se vuelve concreta en una rutina diaria que se repite varias veces al día.

- Actualizar la copia local con el main branch antes de empezar cualquier cambio.
- Trabajar en una rama corta, con un alcance que se pueda terminar el mismo día.
- Compilar y ejecutar las pruebas en local antes de empujar, para no ensuciar el main branch con errores evidentes.
- Volver a traer el main branch, resolver los conflictos que hayan aparecido y compilar de nuevo.
- Abrir el pull request y dejar que el pipeline lo verifique en el entorno de referencia.
- Fusionar al main branch el mismo día, una vez que el pipeline está en verde y la revisión de código terminó.

Reglas de protección del main branch

Las políticas del repositorio son lo que impide que la disciplina dependa de la buena memoria del equipo.

- **Pipeline obligatorio:** no se permite fusionar si el pipeline no está en verde.
- **Revisión obligatoria:** al menos una aprobación de otra persona antes del merge.
- **Rama actualizada:** la rama debe estar al día con el main branch antes de fusionarse, para que el pipeline valide la combinación real.
- **Prohibido el push directo:** todo cambio al main branch entra por pull request, incluidos los del líder técnico.

- **Historial lineal:** se prefiere squash o rebase para que el historial del main branch sea legible y bisecable.

Métricas del pipeline

DORA propone medir el pipeline para saber si la práctica funciona o solo existe en el papel.

- **Frecuencia de integración:** cuántas veces al día llega trabajo al main branch por desarrollador.
- **Duración del build:** cuánto tarda la compilación de commit, con diez minutos como techo recomendado.
- **Tasa de builds exitosos:** qué porcentaje de commits produce un build en verde.
- **Tiempo de reparación del build:** cuánto tarda el equipo en devolver el main branch a verde cuando se rompe.
- **Frecuencia de despliegue:** cuántas veces llega software a producción, una de las cuatro métricas DORA de rendimiento de entrega.
- **Tiempo de entrega del cambio:** cuánto pasa desde el commit hasta que ese cambio corre en producción.

Antipatrones frecuentes

La mayoría de los fracasos no vienen de la herramienta, vienen de la disciplina que la rodea.

- **Pruebas inestables:** una prueba que pasa o falla sin que el código cambie destruye la confianza en el pipeline, porque el equipo empieza a reintentar en

lugar de investigar.

- **Ramas de larga vida:** mantener ramas de semanas convierte el merge en un proyecto aparte y elimina el beneficio de integrar seguido.
- **Main branch roto tolerado:** dejar el main branch en rojo mientras se sigue trabajando bloquea a todo el equipo y vuelve el pipeline decorativo.
- **Pipeline lento:** un build de cuarenta minutos deja de ser retroalimentación y se convierte en un obstáculo que la gente aprende a esquivar.
- **Exceso de pruebas de interfaz:** apoyar la regresión en pruebas de UI produce una batería lenta y frágil, que es la fuente más común de inestabilidad.
- **Pruebas dependientes del orden:** pruebas que comparten estado pasan en secuencia y fallan en paralelo, y el error parece aleatorio.
- **CI de fachada:** tener el servidor configurado sin pruebas reales ni integración diaria da la apariencia de la práctica sin ninguno de sus beneficios.

Integración continua, entrega continua y despliegue continuo

Los tres términos se confunden, y describen niveles distintos de automatización.

- **Integración continua:** cada cambio se integra al main branch a diario y se verifica de forma automática. El resultado es un main branch que siempre funciona.
- **Entrega continua:** además de verificar, el pipeline deja listo un artefacto desplegable en cualquier momento. El paso a producción existe, pero lo autoriza una persona.

- **Despliegue continuo:** todo cambio que pasa el pipeline llega a producción sin intervención humana.

La dependencia es estricta. No existe entrega continua sin integración continua, ni despliegue continuo sin entrega continua.

Beneficios

La evidencia de DORA asocia la integración continua con mayor frecuencia de despliegue, sistemas más estables y entregas más rápidas y predecibles.

- Los defectos se detectan minutos después de introducirse, cuando quien los causó todavía recuerda el contexto.
- La integración deja de ser una fase costosa e impredecible y se convierte en un evento rutinario.
- Refactorizar se vuelve seguro, porque la batería de pruebas avisa de inmediato si algo se rompió.
- El main branch se mantiene listo para desplegar, así que la fecha de liberación pasa a ser una decisión de negocio y no un problema técnico.

Requisitos que la práctica impone

Fowler advierte que la integración continua no es gratis ni universal.

- Exige un equipo comprometido y con disciplina de pruebas, porque el pipeline solo vale lo que valen sus pruebas.
- Exige un entorno de compilación reproducible, sin dependencias de la máquina de nadie.

- No encaja bien en proyectos con colaboradores dispersos y sin coordinación, como los proyectos de código abierto clásicos.
- Requiere inversión inicial en automatización, que se paga con el tiempo que deja de perderse en integraciones manuales.

Ejercicios

Ejercicio 1. Configurar un workflow de GitHub Actions para un repositorio propio, que ejecute la batería de pruebas y el análisis estático en cada pull request, y activar la regla de protección que impide fusionar con el pipeline en rojo.

Ejercicio 2. Tomar un proyecto en curso y medir tres cosas durante una semana. Frecuencia de integración al main branch, duración del build y tiempo de reparación cuando el main branch se rompe. Comparar los resultados con los valores recomendados por DORA y proponer una acción concreta de mejora.