Lo sviluppo software: DevOps, SRE e CI/CD

FILOSOFIA DELLO SVILUPPO DEL SOFTWARE

Francesco Gobbi

1 dicembre 2025

La filosofia DevOps

DevOps è una metodologia che promuove la collaborazione tra:

- team di **sviluppo** (*Development*);
- **operatori di sistemi** (*Operations*).

Obiettivo principale:

- rimuovere gli attriti e le barriere tra sviluppo e operations;
- migliorare la consegna del software in modo **continuo**, **affidabile** e **veloce**;
- permettere una più veloce produzione di applicazioni software da rilasciare in maniera continua.

Perché DevOps è sempre più usato

Negli ultimi anni l'approccio DevOps è diventato popolare perché permette di rispondere meglio a:

- mercato in continua evoluzione;
- applicazioni molto complesse, con più moduli che interagiscono;
- basi dati e client eterogenei (desktop, mobile, ecc.).

Risultato per l'azienda:

- consegna del software più **rapida**;
- maggiore aderenza alle esigenze del cliente;
- riduzione dei tempi per avere il **feedback** dagli utenti.

Componenti chiave e strumenti DevOps

Non esistono regole rigide per DevOps, ma linee guida adattate alle diverse organizzazioni.

Tutte, però, hanno due componenti comuni:

- **Continuous Integration (CI)**: integrazione frequente dei cambiamenti e aggiornamenti del codice;
- **Continuous Deployment (CD)**: automazione del processo di rilascio del software man mano che una nuova release è disponibile.

Componenti chiave e strumenti DevOps

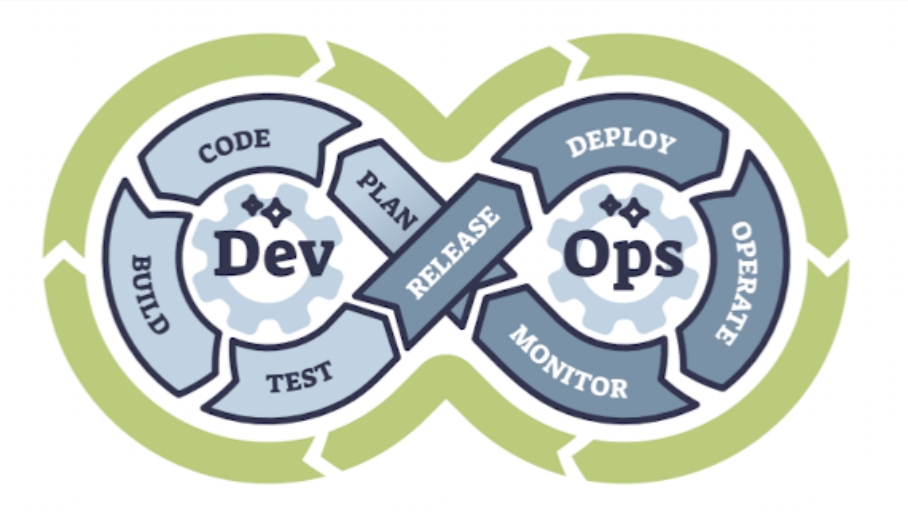
Fasi del ciclo DevOps e perché sono importanti:

1. **Pianificazione (Plan)** – definisce requisiti, obiettivi e priorità del progetto.
   Esempi: Jira, Trello, Azure Boards.
2. **Sviluppo (Develop)** – scrittura del codice e versionamento collaborativo.
   Esempi: Git, GitHub, GitLab.
3. **Build e integrazione continua (CI)** – automatizza la compilazione, i test e l'integrazione nel ramo principale.
   Esempi: Jenkins, GitHub Actions, GitLab CI/CD.
4. **Test** – verifica che il software sia corretto, stabile e sicuro.
   Esempi: Selenium, JUnit, Postman.
5. **Rilascio continuo (Release)** – prepara e rende disponibili nuove versioni, riducendo il rischio di errori.
   Esempi: Jenkins, AWS CodeDeploy, Canary Release.

Componenti chiave e strumenti DevOps

6. **Deployment (Deploy)** – distribuisce automaticamente il software negli ambienti di test, staging e produzione.
   Esempi: Kubernetes, Docker, Ansible.
7. **Monitoraggio e operazioni** – raccoglie metriche, log e performance per rilevare anomalie e problemi.
   Esempi: Prometheus, Grafana, ELK Stack, Datadog.
8. **Report, statistiche e analytics** – analizza i dati per valutare qualità, affidabilità e trend del sistema.
   Esempi: Nagios, Allure, Zenoss, dynatrace, Reportal.io.
9. **Feedback e ottimizzazione (Feedback & Improve)** – utilizza i dati di utenti e monitoraggio per migliorare prodotto e processi.
   Esempi: strumenti di raccolta feedback e metriche interne.

Fasi DevOps


CODE
PLAN
BUILD
TEST
RELEASE
DEPLOY
OPERATE
MONITOR
Dev
Ops

Collaborazione tra Development e Operations

La collaborazione Dev–Ops consente di migliorare qualità e velocità dello sviluppo:

- **Coinvolgimento precoce**: sviluppatori e operatori partecipano fin dall'inizio, condividendo esigenze e obiettivi.
- **Individuazione rapida dei problemi**: feedback continuo tra i team.
- **Riduzione dei tempi di consegna**: risposta più veloce alle richieste del mercato.
- **Migliore qualità del software**: bug individuati prima del rilascio, meno ticket e meno frustrazione.
- **Cambiamento culturale**: serve un ambiente **aperto e collaborativo** e una comunicazione efficace.
- **Sicurezza e best practice**: il software deve rispettare normative e standard aziendali (compliance).

Sicurezza e best practice

Nel mondo DevOps non vanno trascurati:

- i problemi di **sicurezza** del software, soprattutto al momento del rilascio;
- la conformità a **normative** e **standard** del settore (compliance);
- l'adozione di **best practice** aziendali documentate e ripetibili.

Site Reliability Engineering (SRE)

Lo **SRE** è un approccio che unisce competenze di sviluppo e amministrazione dei sistemi per garantire:

- **affidabilità** e **sicurezza** dei servizi online;
- automazione delle attività operative;
- monitoraggio continuo e intervento rapido sui problemi.

Punti essenziali:

1. **Affidabilità del sito**: servizi sempre funzionanti e stabili.
2. **Automazione**: riduzione delle attività manuali e monitoraggio automatico.
3. **Gestione degli incidenti**: analisi cause e ripristino veloce.
4. **Scalabilità**: sistemi in grado di gestire carichi crescenti.
5. **Sicurezza**: protezione dati e conformità a norme e policy.

Introduzione alla CI/CD

La **CI/CD** (Continuous Integration / Continuous Delivery) è una metodologia che permette di rilasciare **aggiornamenti del codice** in modo continuo, frequente e sicuro.

- Integra una "**cultura**" basata su pratiche automatizzate.
- Automatizza integrazione, test, distribuzione e monitoraggio.
- Supporta metodologie **Agile** e approcci **DevOps/SRE**.

La pipeline CI/CD

La **pipeline** è il sistema automatizzato che gestisce lo sviluppo, i test e la distribuzione delle nuove versioni del software.

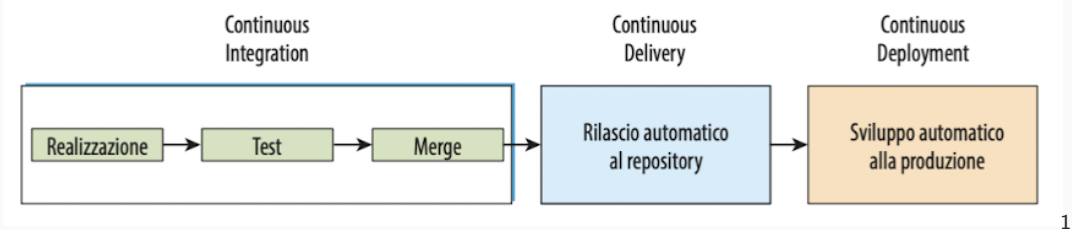
Caratteristiche:

- sequenza di passaggi standardizzati e automatizzati;
- sviluppo tramite **piccoli cambiamenti** controllati;
- versionamento frequente del codice (Git, GitHub);
- riduzione del rischio di errore grazie ai test automatici.

Dove si ferma la CI e dove inizia la CD

Nella pipeline CI/CD:

- la **Continuous Integration** riguarda l'integrazione del codice nel ramo principale e la sua verifica automatica;
- la **Continuous Delivery** interviene quando la CI termina con successo e prepara il rilascio automatico;
- talvolta si parla anche di **Continuous Deployment**, cioè rilascio automatico fino alla produzione.



Esempio di pratica CI/CD in un team

In molti team esiste uno **standard giornaliero minimo di commit** del codice.

- Ogni sviluppatore carica spesso il proprio lavoro nel repository con le varie versioni (es. GitHub).
- Il server di integrazione continua controlla automaticamente i commit.
- Le modifiche vengono testate con unit test e test di integrazione.

Scopo: rendere il codice nel ramo principale una base stabile su cui tutti possono lavorare.

Obiettivi della CI/CD

Ogni team mira a consegnare applicazioni di qualità seguendo due azioni chiave:

1. **Testing continuo**: esecuzione automatica di test di regressione, test di performance e altri test.
2. **Automazione della delivery**: aggiornamento automatico delle applicazioni nei vari ambienti (sviluppo, test, produzione).

I tool CI/CD semplificano la gestione dei parametri legati a ciascun ambiente e lavorano a stretto contatto con gli strumenti di controllo di versione come GitHub.

L'integrazione continua: flussi di lavoro CI

A seconda degli strumenti e del progetto, i **flussi di lavoro CI** possono variare, ma hanno in comune cinque passaggi fondamentali:

1. invio al **repository** del nuovo codice scritto e/o aggiornato;
2. **analisi statica** del codice;
3. **test pre-implementazione**;
4. creazione di **pacchetti** e distribuzione nell'ambiente di test;
5. **test post-implementazione**.

Modalità operative della CI

In generale esistono due modalità operative per l'integrazione:

- **Rilascio periodico giornaliero**: a orari prestabiliti viene effettuato il passaggio dal ramo di sviluppo al ramo *master*.
- **Rilascio in particolari giorni** (*merge day*): integrazione di molte modifiche accumulate; operazione più complessa e rischiosa.

Per ridurre i problemi di compatibilità è preferibile una CI con **commit frequenti**, cioè con interazioni ravvicinate tra sviluppatori.

Dopo ogni integrazione avviene la **convalida**: batterie di test automatici verificano il codice ai vari livelli (unità e integrazione); se emergono problemi di incompatibilità si interviene subito con la correzione.

Distribuzione continua e/o deployment continuo

L'acronimo **CD** può indicare:

- **distribuzione continua**;
- **deployment continuo**.

Sono concetti simili e spesso usati in modo intercambiabile, ma legati a fasi diverse della pipeline:

- la **distribuzione continua** automatizza il rilascio del codice convalidato su un repository (ad es. GitHub o registro per container);
- il **deployment continuo** automatizza il passaggio dal repository all'ambiente di produzione, rendendo l'applicazione subito utilizzabile.

Distribuzione continua e deployment: vantaggi e costi

Distribuzione continua:

- ogni passaggio dopo la CI (build, test, caricamento nel repository) è automatizzato;
- si ha sempre un **nuovo codice pronto al deployment**;
- consente di rilasciare applicazioni tramite **piccole integrazioni** invece che grandi aggiornamenti.

Deployment continuo:

- il rilascio in produzione è automatico quando tutti i test sono superati;
- è meno rischioso rilasciare piccole modifiche frequenti che grandi release uniche.

Contro: richiede un investimento iniziale significativo per progettare test automatici completi e compatibili con tutte le fasi della CI/CD pipeline.

La gestione delle funzionalità

Le principali tecniche per la gestione delle funzionalità sono:

1. **Flag di funzionalità**:
   - permettono di attivare o disattivare parti di codice in fase di esecuzione;
   - le funzionalità in sviluppo hanno un flag che le "disattiva" finché non sono pronte;
   - strumenti tipici: CloudBees Rollout, Optimizely Rollout, LaunchDarkly.
2. **Branching del controllo di versione**:
   - realizzato tramite modelli come **GitFlow**;
   - prevede due branch principali (*master* e *develop*) e branch di funzionalità dedicati;
   - permette di sviluppare nuove feature senza interferire subito col codice stabile.

Packaging, test automatizzati

Nel **packaging** della CI/CD:

- si confeziona l'applicazione in un pacchetto pronto per l'installazione in produzione o in test;
- il pacchetto include eseguibile, database, librerie, risorse e script di configurazione;
- le build possono essere avviate **on demand** a seguito di commit nel repository.

Durante e dopo il packaging vengono eseguiti **test di regressione**:

- verificano che le nuove modifiche non abbiano rotto funzionalità esistenti;
- si basano su **framework di prova automatizzati**;
- possono includere test di prestazioni, test API, analisi statica e test di sicurezza.

Benefici CI/CD

Benefici del metodo CI/CD:

- riduce la complessità nella gestione del codice e migliora l'efficienza dei flussi di lavoro;
- diminuisce il rischio di introdurre bug nei rilasci;
- migliora l'**esperienza utente** grazie ad aggiornamenti più frequenti ed efficaci.

Conclusioni

- **DevOps** è un approccio che combina collaborazione e automazione tra team di sviluppo e operations per migliorare efficienza e qualità del processo di sviluppo e distribuzione del software.
- **CI/CD** si concentra sulla gestione del **ciclo di vita del software**: dall'integrazione continua alla consegna e al deployment.
- Entrambi i metodi puntano all'**automazione** dei processi di integrazione del codice, accelerando il passaggio dall'idea al rilascio in produzione, dove il software genera valore per l'utente.
- La CI/CD è una **componente chiave** del DevOps: è l'insieme di pratiche e strumenti che supportano il flusso di lavoro di sviluppo e distribuzione in un ambiente DevOps, affiancando le tecniche che aumentano la collaborazione tra team di sviluppo e team operativi.