Appunti su Dev Ops e Continuous Integration

Belliato Riccardo

8-14 Aprile 2021

1 L'esperimento in Microsoft (2008) [5]

Team di sviluppo di piccole-medie dimensioni (una dozzina di persone), ma distribuite in giro per il mondo, quindi molto difficile applicare alcuni paradigmi dell'Extreme Programming come il Pair Programming e lo Shared Code Ownership. In media veniva svolta un check-in al giorno. Ad ogni check-in il server compilava, testava ed effettuava l'analisi statica del codice. Lo stesso server compilava l'installer MSI per testarlo una o due volte al giorno, più una ulteriore build notturna per raccogliere dati e metriche. La suite utilizzata è Team Foundation Server su una macchina virtuale con Windows Server 2003.

In totale sono state svolte 551 check-in, di queste 69 sono fallite. L'analisi statica del codice è stata la causa più frequente di fallimento (il 40%) e il tempo necessario a correggere un failure per la quasi totalità dei casi (50 check-in) è di circa 20 minuti. I problemi maggiori si sono riscontrati quando il server andava in crash, spesso durante la build notturna, per mancanza di risorse hardware.

Sono da segnalare le builds fallite in fase di compilazione: spesso la causa di queste erano per errori degli sviluppatori (impostazioni errate, file mancanti,...) segno della iniziale poca familiarità con lo strumento.

1.1 Conclusioni

L'autore dai dati raccolti ha stimato un overhead dovuto al processo di CI (gestione server, fix dei breaks,...) di 267 ore. L'alternativa più credibile era quella di impostare autonomamente, da parte dei singoli sviluppatori, l'ambiente di test ad ogni check-in. Per questo processo, assumendo che tutti i check-in fossero andati a buon fine, si è stimato un overhead di 464 ore.

Personalmente, da questo articolo ho tratto le seguenti conclusioni:

- I vantaggi maggiori da un approccio di sviluppo CI-oriented si hanno sul medio/lungo periodo
- Sul breve gli ostacoli maggiori sono rappresentati dalla fase di formazione del team e di impostazione dei server
  - Quest'ultimo punto in particolare si è rivelato particolarmente critico nell'esperienza descritta, da qui la necessità di avere a disposizione ambienti di CI facilmente scalabili
- Tra questi il vantaggio principale è quello di gestire separatamente e in modo concorrente la parte di sviluppo e quella di compilazione/test/analisi, laddove un approccio tradizionale richiederebbe di interrompere momentaneamente lo sviluppo.

L'articolo si conclude con delle considerazioni riguardanti l'applicazione dei metodi agili in presenza di team di sviluppo sparsi in tutto il mondo, più che sull'utilizzo di metodi di CI in sè

2 Continuous practices [7]

2.1 Definizioni

Definiamo come *continuos practises* (CP) le pratiche di sviluppo che permettono il rilascio rapido e frequente di features e prodotti. Nell'ordine

- **Continuous Integrations (CI):** pratica di sviluppo nel quale i membri del team integrano i cambiamenti nel codice frequentemente. Include anche il building e il testing automatico del software. La CI è considerata il primo step di adozione delle continuous practices.
- **Continuous DElivery (CDE):** insieme di pratiche e processi con lo scopo di assicurarsi che un'applicazione sia sempre pronta ad essere rilasciata in produzione dopo aver superato i test e i controlli di qualità
- **Continuos Deployment (CD):** simile al CDE, con la differenza che il CD è un processo *automatico push-based* (il delivery è provocato dai cambiamenti nel codice), mentre il CDE è un processo *manuale pull-based* (avviene in seguito a una richiesta esplicita). Di conseguenza possiamo considerare il CD come un CDE continuato, che, a differenza del CDE, non può essere applicato a tutti i contesti. Su questa definizione, tuttavia, è ancora in corso il dibattito accademico e industriale.

Queste tre pratiche vengono inquadrate nell'ambito del **continuous software engineering**, branca emergente dell'Ingegneria del Software che si occupa di inquadrare le CP all'interno di tre fasi distinte:

1. Business Strategy and Planning
2. Development
3. Operations

.

2.2 Approcci e strumenti per facilitare le CP

Per supportare e facilitare le CP l'articolo [7] individua 30 tra approcci e strumenti suddivisi in 6 categorie non-mutualmente esclusive:

1. **Riduzione del tempo di build&test in CI**
2. **Aumentare la visibilità del processo di CI**
3. **Supporto al testing (semi)automatico**
4. **Rilevare violazioni, difetti ed errori nel processo di CI**
5. **Risolvere i problemi all'interno delle pipelines di deployment**

6. **Aumentare la dependability e la reliability nel processo di deployment**

.

Le tecniche analizzate sono le seguenti (tra graffe la/e categoria/e a cui appartiene):

- Tecniche di Wallboarding {2} - Indica lo status di integrazione di ogni branch del progetto
- SQA-Mashup {2}- Framework per integrare e visualizzare i dati prodotti in ambienti CI
- VMVW/VMVMVM (Virtual Machine in Virtual Machine/Virtual Machine in a Virtual Machine on a Virtual Machine) {1} - Permettono di isolare le dipendenze tra i vari casi di test e di eseguirli in parallelo
- Approcci di gestione degli errori di reliability in ambienti cloud {6}
- CIViT (Continuous Integration Visualization Tecnique) {2} - consente ai membri del team di non duplicare gli sforzi di test e comprendere visivamente lo stato (cioè, tempo ed estensione) della verifica degli attributi di qualità
- WECODE {4} - Rilevazione di potenziali conflitti di merge in tempo reale
- uBuild {1, 3} - creazione di test riproducibili e deterministici
- Backtracking incremental CI {4}
- BuildBot Robot {2} - Individua, in caso di failure, chi ha caricato il codice difettoso e lo notifica a tutto il team di sviluppo
- Hydra {1} - Tool di building basato su Nix che riproduce automaticamente gli ambienti di building per i progetti
- SQA-Profile {2} - Composizione dinamica delle CI dashboard sulla base di una serie di regole e delle attività degli stakeholders dell'ambiente di CI (per esempio, il project manager)
- Hotspot Approach {1, 4} - Individua i file di header che fanno da bottleneck per il processo di build
- Secure Build Server {5} - isola i build jobs tra loro in ambienti virtualizzati
- Individuazione delle feature difettose in base ai test automatici di interazione {3, 4} - Se i test per un prodotto contenente la feature A e la feature B falliscono, allora almeno una delle due feature non funziona e viene testata singolarmente. Viceversa se i test vanno a buon fine possiamo assumere che le feature funzionano entrambe.

- Aumentare la conoscenza dei membri del team rispetto allo stato delle build {2, 4}
- Tecniche di fault localization e assegnazione di priorità ai test {1, 4}-
- NHN Test Automation Framework {2, 3} - rappresentazione dei risultati di test e gli ambienti attraverso tabelle
- CI Testing for SOA {3} - Utilizzo di framework di test unificati (UTF) per la generazione semi-automatica dei test tramite l'uso di diagrammi di sequenza come input nell'ambito delle SOA (Service-Oriented Application)
- User-defined Script {4} - Convenzioni e regole di qualità del codice da rispettare e controllate in fase di pre-commit
- Enterprise CI {5} - estensione del normale CI. Integrazione continua di moduli interdipendenti testati singolarmente
- Tinderbox {2, 3, 4} - Sistema di testing e di CI con visualizzazione dei risultati su pagine web e sistema di notifica [6]
- Surrogate {3, 4} - Framework di simulazione per gli integration testing di SOA
- CiCUTS {3, 4} - Integra CUTS, system modeling execution tool, con la CI. Permette di emulare diversi ambienti di esecuzione per individuare eventuali problemi di performance
- Continuous Regression Test Selection {1} - Seleziona automaticamente i regression tests da eseguire in base ai cambiamenti nel codice
- Continuous Test suite Prioritization {1, 4}
- Rondo {6} - Tool automatico di CD/CDE in ambienti dinamici, ad esempio il pervasive computing
- Code-Churn Based Test Selection {1, 2} - analizza il code churn (la "misura" di quanto una o più parti di codice sono cambiate) e seleziona i test da eseguire in base a questa
- Miglioramento della sicurezza delle pipeline di Deployment {5}
- Morpheus {1, 2} - Ogni sviluppatore riceve solo i risultati di test riguardanti il codice cha ha modificato ed esegue i test in un ambiente il più simile possibile a quello di produzione
- Process Oriented Dependability {6} - Aumenta la dependability del processo di deployment in ambienti cloud. Modella le operazioni sporadiche come processi, in modo da facilitare le diagnosi di errore all'interno di questi quando accadono.

2.3 Strumenti principali per l'implementazione delle pipelines

La parte più critica dell'implementazione di una deployment pipeline è l'automazione di questa per due motivi principali:

1. Alcune task manuali non possono essere evitate (ad esempio quelle di quality assurance),
2. Attualmente non esistono degli standard

La letteratura divide le pipelines in sette stadi (non tutti obbligatori, solitamente ci si limita ai primi cinque), per ognuno di questi verranno elencati i alcuni tra i principali strumenti e software utilizzati:

1. Sistemi di controllo versione (Git, SVN): in questo stadio gli sviluppatori effettuano il push del codice su repository remoti (es. GitHub)
2. Code management e strumenti di analisi (SonarQube): analizzano e calcolano delle metriche di qualità del codice.
3. Build system (Make, Maven, MSBuild): tool automatici di compilazione e pacchettizzazione del SW
4. Continuous Integration Server (Jenkins, Azure DevOps, GitHub, Gitlab CI): controllano il repository di codice ed eseguono i tool automatici di building del SW. Alcuni integrano anche gli strumenti di code management e di effettuare il deploy del software senza passare per ulteriori server.
5. Tool di testing (NUnit, JUnit)
6. Configuration and Provisioning (Puppet, Yum, Chef, Docker): sistemi di configurazione degli ambienti di testing e deploy
7. CD/CDE Server (Jenkins, HockeyApp, Octopus Deploy)

2.4 Ostacoli

2.4.1 Ostacoli all'adozione delle CP

- Problemi di comunicazione e team awareness
  - Mancanza di awareness (conoscenza del team e delle attività) e di trasparenza. Poca conoscenza dei cambiamenti nel main branch porta a un incremento di merge conficts
  - L'adozione delle CP richiede molta collaborazione e coordinazione tra tutti i membri del team rispetto ad altre metodologie di sviluppo
- Mancanza di investimenti

  - Adottare le CP comporta un aumento dei costi, non solo per le infrastrutture, ma anche per la formazione dei membri dei team
  - Le CP richiedono la presenza di figure professionali altamente specializzate in un settore relativamente nuovo.
  - Si è visto in alcuni contesti che la transizione alle CP ha portato a un aumento dello stress e a una diminuzione della qualità del codice prodotto. Un'ipotesi per spiegare questo fenomeno è che alcuni (soprattutto i più esperti) sentano la pressione di essere ritenuti direttamente responsabili dell'esperienza utente del cliente.
  - Uno dei maggiori ostacoli all'adozione delle CP sono, paradossalmente, gli strumenti e i software presenti sul mercato, i quali presentano spesso problemi di sicurezza ed affidabilità, non sono pienamente compatibili tra di loro (soprattutto in ambito cloud) e cambiano troppo tra una release e l'altra.

- Resistenza ai cambiamenti
  - Cambiare non è mai facile, soprattutto se questo richiede investimenti e tempo. Da non sottovalutare anche un generale scetticismo rispetto alle CP dovuto alla paura di esporre codice qualitativamente scadente e di aumentare lo stress e la pressione nell'ambiente di lavoro dovuto ai rilasci continui, oppure di non essere in grado di capire quale release causa problemi

- Processi organizzativi, strutturali e politici
  - Difficoltà nel cambiare culture e politiche lavorative consolidate.
  - L'adozione delle pratiche di CI è particolarmente difficoltosa nell'ambito dello sviluppo distribuito perchè quest'ultimo diminusce la visibilità del progetto

2.4.2 Ostacoli all'adozione del CI

- Testing
  - Una strategia di testing poco strutturata e una qualità dei casi di test scadente, nonostante non impediscano l'utilizzo delle pratiche di CI, ne riducono enormememente l'efficacia e la confidenza degli sviluppatori con lo strumento. Ad esempio, test troppo lunghi e complessi impediscono di ricevere il feedback immediato per poter intervenire tempestivamente in caso di errori

- Merging conflicts
  - Architetture con elevato accoppiamento tra moduli, poca visibilità del progetto e incompatibilità tra dipendenze aumentano il numero di merge conflicts e quindi, i tempi necessari all'integrazione delle funzionalità

2.4.3 Ostacoli all'adozione della CDE

- Architetture inadatte
  - Software monolitici, con molte dipendenze ed elevato accoppiamento tra moduli comportano dipendenze elevate tra i team di sviluppo oltre ad amplificare enormemente l'effetto dei cambiamenti nel codice
  - Cambiamenti troppo frequenti ai database comportano un aumento del tempo necessario a integrate e testare il SW (ad ogni integrazione bisogna rigenerare il database) con conseguenti bottleneck nello sviluppo
- Dipendenze all'interno del team di sviluppo
  - Team di sviluppo troppo interdipendenti impediscono ai singoli team di poter sviluppare, testare, integrare e rilasciare le applicazioni in modo indipendente dagli altri. Può essere conseguenza di architetture sw inadatte.

2.4.4 Ostacoli all'adozione della CD

- Clienti
  - Ogni cliente opera all'interno di un contesto ben definito, spesso complesso e configurato manualmente e di cui spesso si hanno poche informazioni. Risulta difficile stabilire delle configurazioni automatiche di rilascio per ogni cliente e per ogni versione.
  - La presenza di applicazioni e software legacy richiede lo sforzo aggiuntivo di dover testare, a ogni release, che non ci siano problemi di integrazione coi SW esistenti
  - Non tutti i clienti vedono di buon occhio i rilasci frequenti (notifiche di aggiornamento moleste, incremento dei bugs nel SW, plug-in incompatibili tra una release e l'altra,...)
- Dominio
  - Non tutti i domini sono adatti ai rilasci frequenti, in particolare quelli safety critical.

2.5 Pratiche comuni

2.5.1 Pratiche comuni di implementazione delle CP

- Miglioramento della comunicazione e della visibilità del team di sviluppo
- Aumento degli investimenti
  - Migliore documentazione e pianificazione del lavoro. Utilizzo di Modelli di Integrazione descrittivi e standardizzati per aumentare la visibilità del progetto.

  - Miglior coinvolgimento del team di sviluppo, con conseguente minor stress
  - Aumento del livello di formazione e di esperienza medio del team
- Cambiamenti nella struttura lavorativa
  - Definizione di nuove figure professionali come il release manager oppure il "build sheriff", ossia colui che si occupa di controllare costantemente lo stato dei processi di build e di risolvere i problemi senza coinvolgere gli sviluppatori
  - Adozione di nuove regole e politiche aziendali. Ad esempio: politica di rotazione dei ruoli tra membri del team oppure limiti di tempo per risolvere un errore, pena il rifiuto del commit.

2.5.2 Pratiche comuni di implementazione della CI

- Miglioramento dell'attività di testing
  - Adozione del test-driven development
  - Redazione del test planning tra il team di controllo qualità e gli sviluppatori
  - "Cross-team testing": l'integration testing del modulo A viene eseguito da un team che non è stato coinvolto dall'implementazione di A
- Implementazione di strategie di branching
- Scomposizione della fase di sviluppo in piccole unità
  - Necessario per mantenere basso il tempo di build&testing. Una proposta in questo senso è stata fatto dividendo la fase di sviluppo in tre blocchi:
    1. Interface Module
    2. Platform Independent Module
    3. Native Module

2.5.3 Pratiche comuni di implementazione della CDE

- Utilizzo di architetture flessibili e modulari in modo da essere sviluppate e rilasciate in modo indipendente. Molti studi confermano che la maggior parte delle decisioni e dei cambiamenti atti a implementare la CDE sono architetturali. Importante anche definire chiaramente le interfacce delle componenti.
- Coinvolgere tutti nel processo di deployment

2.5.4 Pratiche comuni di implementazione della CD

- Release parziali: alcune pratiche
  - Rilascio limitato solo a certi utenti (canary deployment)
  - Nascondere e disabilitare le funzionalità nuove (o problematiche) agli utenti (dark deployment)
  - Rolling back immediato alle versioni stabili
  - "empty release": il team di sviluppo modifica e incrementa le pipeline di volta in volta a partire da una molto semplice (es. Hello World)
- Coinvolgimento del cliente: due approcci
  - "lead customer": i clienti, oltre a far parte del processo di sviluppo, vengono istruiti alle pratiche e ai concetti della CD
  - "triage meeting": il cliente organizza i meetings con gli sviluppatori e decide i cambiamenti e le priorità dei requisiti.
  - In generale, il coinvolgimento del cliente è importante per l'adozione della CD/CDE, in particolare per quelle aziende che non hanno i mezzi sufficienti per adottare la CD

3 DevOps

Le definizioni di DevOps sono molteplici in letteratura, l'articolo [3] riporta la seguente:

> DevOps (Development and Operations) è uno sforzo collaborativo e interdisciplinare all'interno di un'organizzazione per automatizzare il rilascio continuo di nuove versioni del software, garantendo allo stesso tempo correttezza e affidabilità.

Più in generale DevOps possiamo considerarlo come una "branca emergente dell'ingegneria del SW e una filosofia che utilizza dei team cross-funzionali per buildare, testare e rilasciare SW velocemente e in maniera affidabile in modo automatico" [4] e una naturale evoluzione dei metodi agili, i quali non hanno mai messo troppa enfasi sul processo di deployment del software [3]. Lo scopo principale della filosofia DevOps è quella di colmare il gap tra i team di sviluppo e quelli di IT Ops [4].

Questo scopo può essere visto su tre punti di vista differenti [3]:

- **Ingegneri** ai quali interessa:
  - qualificarsi e formarsi ai metodi e alla filosofia del DevOps attraverso la numerosa letteratura scientifica fin qui pubblicata
  - ristrutturare le architetture SW in modo che siano adatte alla CI
- **Managers** interessati a:
  - capire come introdurre il DevOps all'interno delle loro organizzazioni
  - come misurare la qualità delle pratiche di DevOps adottate
- **Ricercatori e accademici**:
  - condurre studi sullo stato dell'arte del DevOps e contribuire alla discussione sul tema con ingegneri e managers
  - educare i nuovi ingegneri alle pratiche di DevOps

Per riempire questo "gap" nella letteratura sono stati proposti tre possibili approcci [4]:

- Mix Responsabilities: gli ingegneri hanno la responsabilità sia dello sviluppo che delle operazioni
- Mix personnel: aumentare la comunicazione tra Dev e Ops (mantenendo però i ruoli separati)
- Bridge team: introdurre un DevOps team separato con il compito di fare da ponte tra Dev e Ops

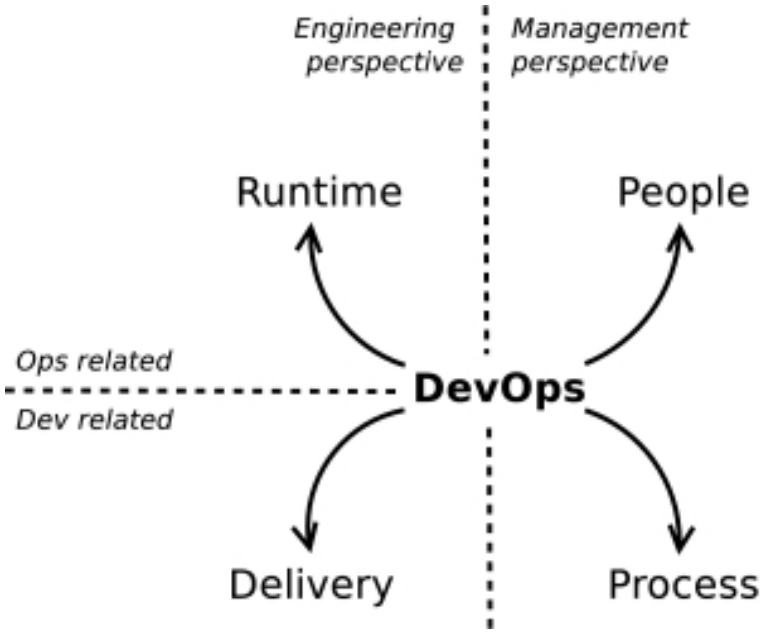

Figura 1: Concetti e prospettive di DevOps - immagine tratta da [3]

3.1 Concetti fondamentali [3]

- **Process** Uno dei principali risultati che DevOps mira a raggiungere è quello di ridurre rischi e costi, aumentare la qualità dei prodotti sw e la soddisfazione del cliente, attraverso release frequenti e affidabili. Per raggiungere questi scopi la filosofia DevOps promuove l'utilizzo di metodi agili e cicli di feedback molto brevi.

- **People** Come già detto in precedenza, lo scopo principale di DevOps è quello di "abbattere" la normale distinzione tra team di sviluppo e team di IT. Il problema sta nel come questo debba essere fatto e che cosa vuol dire essere un operatore DevOps. In questo contesto sono state proposte due visioni differenti [4]:

  – "DevOps as culture": la conoscenza è condivisa tra sviluppatori e DevOps, per cui questi ultimi conoscono il codice e aiutano i primi a risolvere i problemi. Inoltre i DevOps hanno il compito di scrivere le user stories e distribuire le tasks.

  – "DevOps as job descriptions": compito dei DevOps è quello di mettere a disposizione degli sviluppatori le infrastrutture (pipelines, server di deployment,...).

- **Delivery** Parte centrale del processo di DevOps è l'adozione delle CP (vedi sezione 2), tuttavia l'utilizzo di queste non implica necessariamente

anche l'adozione della filosofia. Uno degli ambiti di maggior successo di DevOps è la creazione e il versionamento di microservizi ed API (vedi sezione 3.3). Il concetto di *Delivery* è quello più universalmente accettato e consolidato.

- **Runtime** Oltre a rilasciare velocemente le nuove versioni è altrettanto importante che queste siano affidabili e stabili, motivo per cui due delle caratteristiche principali di DevOps sono il monitoraggio costante delle metriche, sia di business (feedback) che di più basso livello (codice, infrastruttura, test,...), e quella di poter effettuare esperimenti direttamente in produzione, con le tecniche di CD viste in sezione 2.5.4

3.2 Toolset

DevOps condivide buona parte degli strumenti e suite software con quelli già in uso nel contesto delle CP, già elencati in sezione 2.3. A questi aggiungiamo altre due categorie di toolset specifici per il DevOps:

- Knowledge Sharing di cui fanno parte le wiki, le documentazioni e i sistemi di messaggistica (istantanea e non).
- Monitor&Logging, ossia quelle applicazioni con il compito di tracciare tutti gli aspetti non-funzionali delle applicazioni come affidabilità, scalabilità, performance.

3.3 Approfondimento sulle architetture CP-compliance: microservizi ed API [3]

Si è già discusso dell'importanza delle scelte architetturali nell'adozione delle CP (vedi 2.5.3). Uno degli ambiti applicativi particolarmente adatti all'adozione delle CP e del DevOps è la creazione e l'implementazione di architetture a microservizi. Più questi sono indipendenti tra loro e presentano un basso livello di accoppiamento con altri microservizi più questi sono testabili ed è facile effettuare rapidamente il deploy, esattamente i requisiti ricercati per la CI/CD. Altro punto a favore delle API è quello di adattarsi efficamente e rapidamente in contesti fortemente regolamentati [3]. Tuttavia presentano anche questi alcuni problemi:

- Eterogeneità dei microservizi, soprattutto quando riguardano aspetti non-funzionali (logging, startup, configurazioni,...)
- Le versioni rilasciate devono essere le stesse utilizzate in fase di integration testing, pena il rischio che queste non funzionino correttamente per incompatibilità.

A questo proposito la letteratura ha proposto alcuni pattern per risolvere questi problemi:

- Standardizzare un piccolo set fondamentale di microservizi all'interno dell'organizzazione
- Utilizzare una pipelines diversa per ogni microservizio
- Utilizzare log aggregators, registri per i servizi, ID di correlazione
- Segregare codice sorgente, configurazioni e variabili d'ambiente
- Utilizzare sistemi automatici per la scalabilità (load balancers, contratti a consumo, ...)
- Utilizzare sistemi di compatibilità
- Utilizzare sistemi di versionamento della API. Quest'ultimo punto però è controverso, perchè aumenta la difficoltà di deployment e di gestione delle API [2]

3.4 Approfondimento: Infrastructure-as-code (IASC) e standard TOSCA [1]

Il design delle infrastrutture è, come visto in sezione 1 è una parte fondamentale del processo di sviluppo sw, non sono DevOps. Un'infrastruttura inadeguata penalizza fortemente gli aspetti non-funzionali di un SW e può essere causa di fallimento dello stesso. Solitamente, l'installazione e la configurazione di infrastrutture è un'operazione manuale, anche nel caso di installazione in VM. Con lo sviluppo dei metodi DevOps e delle piattaforme cloud si è andata via via promuovendo l'Infrastructure-as-code (IASC), ossia l'utilizzo di nozioni e tecniche tipiche della programmazione per il design delle infrastrutture. L'obiettivo principale di questo approccio è la portabilità attraverso una serie di pratiche come il versioning, l'utilizzo di pattern oppure la specificazione automatica dell'infrastruttura sulla base dell'architettura e del modello del SW. In molte organizzazioni l'utilizzo di IASC è stato un motivo per passare al DevOps.

3.4.1 TOSCA

OASIS TOSCA (Topology and Orchestration Specification for Cloud Applications) è uno standard industriale di IASC e orchestrazione supportato da aziende come HP, CISCO, Cloudera, Huawei, IBM. Di fatto è l'attuale stato dell'arte nell'industria. Si tratta di un linguaggio di modellazione indipendente dalla piattaforma utilizzata e adatto ad applicazioni multi-cloud [1]. TOSCA permette di specificare l'infrastruttura attraverso una topologia a grafo. Nodi e archi di questo grafo sono infrastrutture minime e riutilizzabili descritte attraverso template e sono fortemente tipati. Esempi di orchestratori TOSCA-ready sono Cloudify e Apache Brooklyn (quest'ultimo basato su YAML).

3.5 Implementare DevOps: quattro approcci [4]

Si propongono ora quattro approcci diversi per l'implementazione di DevOps all'interno di una organizzazione:

- *Developers-Ops*: i senior development gestiscono l'infrastruttura automatizzata di CI/CD all'interno di un sistema cloud ibrido. Il team di Ops si occupa di gestire l'infrastruttura fisica presente in sede. Gli altri sviluppatori scrivono il codice, lo rilasciano sulle pipelines messe a disposizione e gestiscono autonomamente le applicazioni. In questo caso il senior development è visto come un facilitatore delle pratiche di DevOps
- *Developes-Outsourced Ops*: simile al Developers-Ops, solo che l'ambiente di deploy è cloud-based, quindi non c'è la necessità del team di Ops
- *Developers-DevOps*: il team di DevOps si occupa di creare e gestire sia le pipelines che l'infrastruttura cloud. Gli sviluppatori si occupano solo di rilasciare il codice
- *DevOps bridge*: Utilizzato in ambienti ibridi. Come nell'approccio Developers-DevOps, gli sviluppatori hanno solo il compito di implementare codice, lasciando la gestione delle pipelines di deploy e delle infrastrutture al team di DevOps. Questi, a loro volta, fanno riferimento al team di IT Ops per quanto riguarda la gestione dell'infrastruttura locale. In questo approccio, ognuno è responsabile delle proprie azioni. Si tratta dell'approccio più utilizzato, anche se molti sono concordi del fatto che non sia l'approccio giusto per implementare DevOps.

3.6 Come qualificare una persona come DevOps Engineer [3]

Una problematica della filosofia DevOps è la formazione. Non esistendo uno standard condiviso, solitamente la formazione si riduce a scegliere uno o più strumenti di DevOps e imparare a usarli, limitandosi spesso a una auto-valutazione. Un approccio più generale può essere quello di insegnare a costruire sistemi scalabili e/o distribuiti con adeguate caratteristiche non-funzionali, tuttavia richiede infrastrutture adeguate e un efficace sistema di valutazione e di correzione e non sempre è possibile farlo in modo automatico. In linea generale si richiede agli aspiranti "DevOps engineers" una certa autonomia e resposabilità e l'attitudine a formarsi in maniera continua. In questo ambito è di vitale importanza l'esperienza di altri colleghi e anche la frequentazione di community e forum specializzati.

3.7 Considerazioni e conclusioni

Il confine tra DevOps e CP è molto sottile, banalmente perchè uno è parte integrante dell'altro e , di fatto, ne condividono obiettivi e gran parte delle problematiche. Le CP nascono e vanno intese come semplici strumenti software

di automazione. DevOps, invece, va intesa più come una filosofia da integrare all'interno di un modello di sviluppo consolidato, come ad esempio l'Extreme Programming. DevOps, sotto certi punti di vista, si è sviluppato per rispondere ad alcune delle problematiche emerse dall'adozione delle CP come:

- la mancanza di comunicazione tra team eterogenei, attraverso l'abbattimento delle barriere tra questi;
- la mancanza di una definizione più generale di ruoli e competenze (che nel caso delle CP erano per lo più specifiche per i singoli contesti d'uso), attraverso le figura del DevOps engineer.

Personalmente, la cosa che ritengo più interessante della filosofia DevOps è l'approccio al ruolo dell'informatico in generale. I metodi di sviluppo classici come il waterfall hanno sempre proposto una rigida divisione dei ruoli all'interno dei team di sviluppo (ingegnere dei requisiti, progettista, sviluppatore, tester, sistemista,...), questo approccio si adatta molto bene a sistemi software molto grandi e complessi con requisiti stabili nel tempo, di contro ha un alto overhead rispetto alla scrittura di codice in sè. DevOps invece propone una sorta di "ritorno alle origini", l'informatico non è più chiamato a specializzarsi e fossilizzarsi su un solo compito, ma deve sapere un po' di tutto tutto: gli sviluppatori devono sapere come funziona il testing e come funziona una pipeline di rilascio, chi si occupa di testing deve essere in grado di mettere mano al codice per effettuare delle correzioni veloci, e così via. La necessità di rilasciare velocemente software a partire dai primi anni '90 ha comportato la necessità di automatizzare il più possibile i processi di sviluppo, da qui lo sviluppo dei metodi agili, delle CP e del DevOps. Da dove nasce questa esigenza di rilasci veloci? Ho formulato due ipotesi:

- Lo sviluppo esponenziale della potenza di calcolo e della complessità dell'hardware a partire dagli anni '80: il rischio era quello di iniziare a sviluppare per un certo hw e rilasciare software legacy già alla prima versione
- Lo sviluppo dei movimenti open-source a partire dagli anni '90, da sempre abituati a integrazioni continue di codice proveniente da singoli o piccoli gruppi eterogenei e a rilasciare versioni intermedie tra una major release e l'altra, hanno costretto l'industria ad adattarsi per motivi di "marketing": perchè acquistare un prodotto software quando esiste un alternativa gratuita e di qualità molto più aggiornato?

Riferimenti bibliografici

[1] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri. Devops: Introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498, 2017.

[2] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[3] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Comput. Surv.*, 52(6), November 2019.

[4] R. W. Macarthy and J. M. Bass. An empirical taxonomy of devops in practice. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 221–228, 2020.

[5] A. Miller. A hundred days of continuous integration. In *Agile 2008 Conference*, pages 289–293, 2008.

[6] Jeremy Nimmer, Brian Fallik, Nick Martin, and John Chapin. Continuous automated testing of sdr software. In *Proceedings of the 2006 Software Defined Radio Technical Conference (SDR 2006), Orlando, Florida (November 2006)*. Citeseer, 2006.

[7] M. Shahin, M. Ali Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.