POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

DevOps e Continuous Delivery

Release-Automation di software mediante implementazione di una pipeline

Relatore
prof. Marco Mezzalama

Candidato
Marco Punzi

Supervisore aziendale
Consoft Sistemi
dott. Marco Casu

Anno accademico 2017 – 2018

Ai complici della
realizzazione di un sogno

Abstract

«L'avvento della Application Economy impone sempre più l'idea del software come strumento per generare direttamente valore di business.

Occorre quindi garantire servizi di deployment rapidi e in alta affidabilità, che permettano il rilascio di software testato e di qualità in qualsiasi istante. La Continuous Delivery quale estensione della Continuous Integration prevede un approccio strutturato ed automatizzato ai test funzionali, di non regressione e di integrazione, ottimizzando i processi ripetibili di Lifecycle e Deploy. Per assicurare soluzioni flessibili ed integrabili all'interno dei processi aziendali, si necessita di un approccio metodologico di gestione del ciclo del software agendo sulla comunicazione e collaborazione tra gli sviluppatori e gli operatori IT.»

(Consoft Sistemi SpA, *DevOps Solution*, 2017)

Indice

Introduzione Generale

L'obiettivo principale di questo lavoro è quello di fornire le linee guida all'implementazione di una soluzione sempre più predominante nell'ambito dello sviluppo e del rilascio di software di elevata qualità - la *Continuous Delivery* - evidenziando i notevoli vantaggi ottenibili utilizzando un approccio di questo genere.

Offrire continuamente servizi migliori - in affidabilità ed efficienza - richiede una sinergia tale da dover rimodellare strutture organizzative monolitiche e logiche di business statiche, in opposizione ad un mercato fortemente dinamico e a tecnologie in continua evoluzione. In questo scenario, il software tende ad abbandonare lo storico ruolo di 'mezzo' per offrire un servizio, identificando esso stesso il servizio e il suo valore economico. Uno dei requisiti fondamentali è quello di predisporre una infrastruttura che implementi e validi il software - continuamente e in maniera automatizzata - minimizzando la presenza dell'intervento manuale, associato troppo spesso a *bottlenecks* temporali nei processi di rilascio e distribuzione di nuove versioni (Figura 1). Il *deployment* continuo, però, è possibile se si adottano strumenti e

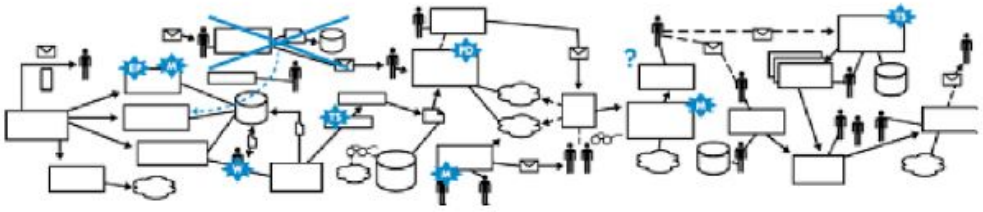

Figura 1. Organizzazione tipica e inefficiente. Rilascio di una nuova versione del software stimato entro alcuni mesi

atteggiamenti mirati a massimizzare la collaborazione tra reparti differenti, affinchè ogni sforzo individuale confluisca verso l'obiettivo comune. Pertanto, verrà posto l'accento su una nuova etica aziendale, incaricata di abbattere le pareti comunicative tra i vari team e che consenta di sincronizzare ogni singola operazione.

Un esempio tipico in questo senso può essere rappresentato dalla Formula 1, palcoscenico in cui l'importanza dei team e la loro efficacia sono sinonimi di successo. Si supponga, in tale scenario, di avere un problema tecnico che rischia di compromettere la partecipazione alla gara a distanza di poche ore, e si provi ad immaginare il danno economico che ne può derivare. Il lasso di tempo a disposizione non concede la possibilità di poter procedere per tentativi, attendendo che nuovo codice venga sviluppato e testato. Il rilascio di una soluzione deve essere immediato, con la probabilità massima di riuscire nella risoluzione del problema. Ecco, allora, che la collaborazione tra i team e il continuo *deployment*, consentono una detection degli errori rapida e un conseguente rilascio di una soluzione adeguata.

L'esempio della Formula 1 rispecchia chiaramente l'esigenza di una innovazione tecnologica continua, centrata sulla collaborazione dei team di sviluppo, al fine di garantire il successo finale. Nei capitoli che seguiranno, verrà innanzitutto contrapposto il modello di sviluppo *Agile* al modello classico di tipo *Waterfall*. In particolare, verranno sottolineati i punti deboli di un modello 'a cascata' non più idoneo alle esigenze di business attuali, introducendo uno sviluppo agile che intende massimizzare l'efficienza riducendo i costi prestazionali e temporali. Per raggiungere tale scopo, però, una nuova etica denominata *DevOps* si rende necessaria per dare vita alla nuova logica implementativa. Nuove dinamiche e nuove metodologie di lavoro hanno il compito di plasmare l'intera infrastruttura organizzativa; si passerà, dunque, a descrivere i princìpi di realizzazione di questo nuovo *asset*, con un approfondimento sul ritorno di investimento alle aziende che decidono di adottarlo.

Dopo aver trattato i princìpi di base, nel secondo capitolo verrà descritta e approfondita la struttura di una pipeline di rilascio del software, la quale, animata dall'etica *DevOps*, si propone di automatizzare l'intero processo di produzione. L'analisi delle funzionalità, relativa a ciascuno stage della pipeline di *deployment*, troverà un riscontro pratico nel terzo capitolo, con l'implementazione di una pipeline reale in esecuzione all'interno di una infrastruttura virtuale creata *on demand* per simularne il comportamento. La *Continuous Integration* non sarà più limitata al *versioning* del codice ma diventerà il motore di una architettura, ben più complessa, in cui il testing e il feedback continuo assumono un ruolo fondamentale. In una prima parte, verrà fornita una implementazione della pipeline tramite la concatenazione di job che eseguono operazioni associate a ciascuna fase; dopodichè, la pipeline verrà gestita tramite uno script, la cui esecuzione in tempo reale andrà a riprodurre il *deployment* e la *release* di una semplice applicazione.

Infine, nell'ultimo capitolo, verranno confrontati i tempi di *release* della stessa applicazione sviluppata e testata prima seguendo l'approccio tradizionale e poi utilizzando la pipeline implementata. Sarà evidente, dunque, come i tempi di rilascio

si riducano dall'ordine dei mesi a quello dei giorni o addirittura ore. Inoltre, verrà riportata la curva di maturità di un processo di *Continuous Delivery* del software. Su di essa sarà evidenziato il livello di ottimizzazione raggiunto dal lavoro proposto e i vari step di ottimizzazione che le varie compagnie sono in grado di raggiungere o in cui si trovano allo stato attuale. Verranno confrontate, quindi, le prestazioni a ciascun livello di avanzamento, sottolineando come sia possibile minimizzare il rischio di eventuali errori in fase di produzione incrementando, allo stesso tempo, la qualità e l'affidabilità del software da rilasciare.

Capitolo 1

Modelli e filosofie a confronto

1.1 Introduzione

Dall'istante in cui il primo computer è stato avviato ad oggi, l'Universo IT ha dovuto fronteggiare una crescente ed insaziabile domanda di applicazioni software e servizi. Nonostante l'affermarsi di innumerevoli tecnologie quali prodotti software *commercial of-the-shelf* (COTS), virtualizzazione, e *cloud computing* abbia cercato di soddisfare tale richiesta, il rilascio di soluzioni software è sempre stato lento e poco efficace. Questo è dovuto, principalmente, a limiti sempre più evidenti in una metodologia di sviluppo e rilascio del software non più adatta alla costante e rapida evoluzione delle necessità dei customers: l'approccio *Waterfall*.

In questo capitolo verranno descritti i princìpi di tale strategia e le fragilità che hanno permesso allo sviluppo *Agile* di emergere e plasmare in maniera efficace delle logiche di business dal volto sempre più informatizzato. In ottica *Agile*, si passerà poi a descrivere le linee di pensiero del *DevOps*, il quale mira a garantire servizi di *deployment* rapidi ed affidabili, che permettano il rilascio di software testato e di qualità in qualsiasi istante.

1.2 Nascita e decadenza del modello *Waterfall*

Il modello *Waterfall*, anche conosciuto come modello a cascata, è stato tra i più utilizzati in ambito *software development*. Il design fortemente sequenziale che lo caratterizza, fa sì che la progettazione risulti in una serie di stadi a cascata, in cui la fase successiva viene eseguita solamente dopo la corretta esecuzione e validazione di quella precedente. Esso fonda le sue origini applicative nell'industria manifatturiera, all'interno delle quali eventuali cambiamenti, a valle di qualsiasi tipo di implementazione su sistemi fisici, risultavano in costi proibitivi oppure in uno stravolgimento massivo delle logiche di produzione. Tuttavia, data l'assenza di metodologie di *software development*, questo approccio *hardware-oriented* è stato inizialmente adattato allo sviluppo del software, nonostante un grado di flessibilità pressoché nullo.

Sebbene il primo utilizzo del termine *Waterfall* risalga al 1976, con la pubblicazione del paper *Software Requirements: Are They Really A Problem?* [1], una prima descrizione formale della metodologia può essere attribuita a Winston W. Royce. Con il suo articolo *Managing the Development of Large Software Systems* [2] pubblicato nel 1970, Royce identifica un pattern incompleto allo stato attuale e con notevoli problematiche, sconsigliandone l'utilizzo e proponendo delle soluzioni per migliorarne il funzionamento. Esistono svariate altre revisioni del modello, le quali culmineranno nella pubblicazione del *manifesto Agile* agli inizi del ventunesimo secolo. Tuttavia, le idee di Royce fanno di lui un pioniere della moderna ingegneria del software. Egli ha permesso una diffusione del modello *Waterfall* nella sua versione più robusta, con occhio lungimirante al cambiamento che avrebbe poi rimosso quello stesso modello da una posizione monopolista nell'ambito di sviluppo e produzione software.

1.2.1 Le fasi del modello

L'articolo di Royce può essere considerato, dunque, una estensione del modello *Waterfall* di base, necessaria a ridurne tempi e costi di sviluppo. Ma quali fasi prevede la metodologia Waterfall nella sua versione più elementare?

Innanzitutto, esistono due step fondamentali e comuni ad ogni tipologia di sviluppo, indipendentemente dalla complessità. Ad una fase iniziale di analisi, segue la fase di implementazione e scrittura del codice (Figura 1.1). Questa semplificazione, però, è indubbiamente inadatta alla gestione software di sistemi complessi. Si rende necessario un approccio più esteso e generale (Figura 1.2), che però si riflette in una implementazione poco funzionale ed esposta a rischi. Il pattern utilizzato è quello di definire inizialmente tutti i requisiti e le caratteristiche dell'applicazione, ad esempio, da sviluppare. Seguendo l'andamento del ciclo di sviluppo, esso culmina in

quello che può essere definito uno step decisionale, la fase di testing. Se, infatti, vengono soddisfatte positivamente le richieste del destinatario del servizio, sia esso un semplice utente piuttosto che un business stakeholder, l'applicazione intraprende il suo ciclo di produzione e manutenzione. In caso contrario, ci si ritrova ad affrontare quanto di negativo questo modello può nascondere in termini di efficienza e costi.

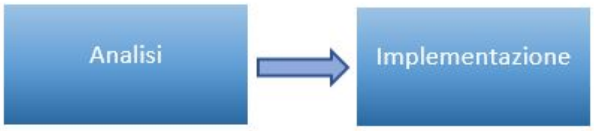

Figura 1.1. Flusso minimale di implementazione

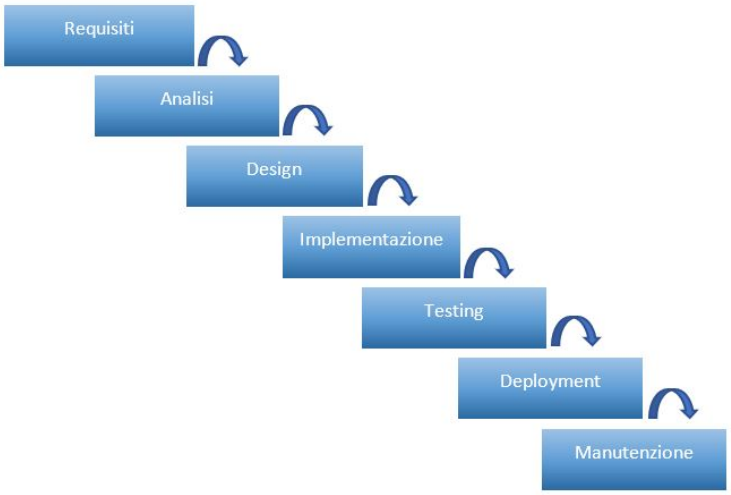

Figura 1.2. Fasi di sviluppo di un programma: versione base del modello a cascata

1.2.2 Gli svantaggi del modello

In un mondo in cui il ruolo dell'*Information and Technology* deve rispondere, offrendo soluzioni rapide ed efficienti a problemi di diverso genere prima che questi vengano pienamente compresi, e in un mondo in cui condizioni e vincoli variano rapidamente in relazione alle esigenze di business, il modello di sviluppo *Waterfall* non figura certamente tra le scelte ottimali. Ciò è dovuto, principalmente, a tre tipologie di trasformazioni:

- *Da prodotto a servizio* – Il consumatore, quotidianamente, predilige la *user experience* che trae dal prodotto rispetto al prodotto fisico stesso
- *Da separazione a fusione* – Non esiste più distinzione tra prodotto fisico e software. Uno smartphone viene visto come un sistema che offre servizi digitali e non più come insieme di circuiti
- *Da semplice efficienza ad agility* – Non è più la differenza di brand a garantire un'ottima reputazione sul mercato. L'abilità oggi consiste nel rispondere rapidamente alle esigenze di mercato, crearne nuove tipologie e incrementare la qualità dei servizi

Considerando questo nuovo scenario, si pone il problema di come le applicazioni vadano progettate, sviluppate e supportate. Tra gli svantaggi del modello *Waterfall* vi è quello di stabilire un set completo di requisiti a monte della fase di progettazione. I fornitori di servizi, al fine di adattarsi al comportamento mutevole del consumatore, rischiano di rilasciare applicazioni cosiddette *white elephant*[1]. Inoltre, nel lasso di tempo impiegato per il rilascio della soluzione, le condizioni di business potrebbero addirittura cambiare e un remake radicale del sistema dovrà essere effettuato. Infine, siccome le applicazioni software vengono rilasciate in maniera massiva successivamente alla fase di testing e *deployment*, la mole delle operazioni IT coinvolte nel supporto e nella manutenzione cresce in maniera significativa.

Il softwarista David Parnas, nel suo saggio *A Rational Design Process: How and Why to Fake It* [3], spiega come alcuni dettagli funzionali di un sistema diventino noti solamente in fase di implementazione. Alcuni requisiti e vincoli, infatti, non possono essere dettati prima che il *lifecycle* dello sviluppo giunga a termine. Per questi motivi, gli aspetti che emergono in itinere potrebbero invalidare il design di progettazione iniziale, costringendo a reiterare il tutto. Idealmente, la reiterazione è confinata a step successivi tra loro (Figura 1.3). Purtroppo, però, ciò non accade e la revisione del design di progettazione rimbalza sulla revisione dei requisiti iniziali (Figura 1.4).

1.3 Lo sviluppo *Agile*

La definizione dei requisiti e dei *constraints* può variare nel tempo, man mano che nuove funzonalità emergono come necessarie. Pertanto, una maggiore flessibilità è richiesta nel *lifecycle* di sviluppo del software, al fine di poter rispondere rapidamente alle nuove necessità. Il modello *Agile* (Figura 1.5) è nato al fine di fronteggiare

[1] «Appellativo che viene dato a beni o servizi i cui eccessivi costi di realizzazione e gestione non siano compensati dai benefici che danno o che potrebbero dare.» (Wikipedia)

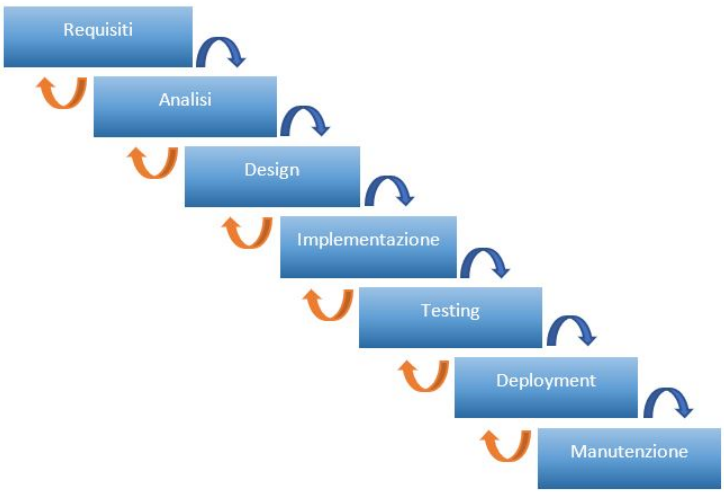

Figura 1.3. Scenario ideale: l'interazione iterativa tra le diverse fasi è confinata a step successivi

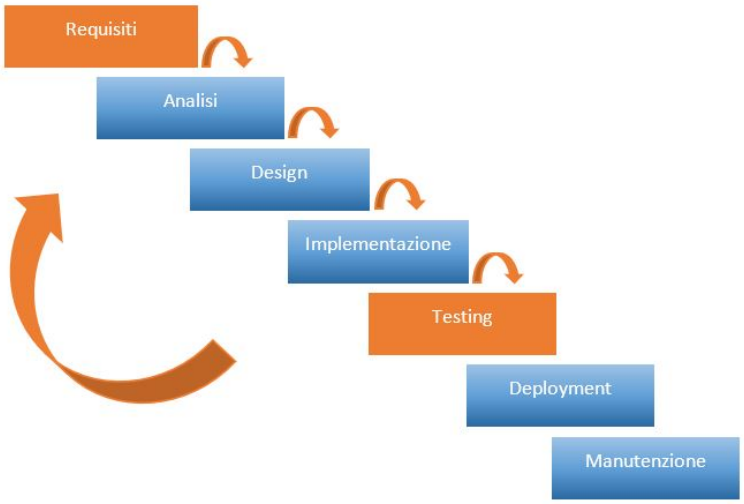

Figura 1.4. Scenario reale: la reiterazione del processo non è confinato a step successivi

queste nuove sfide. Il *Manifesto for Agile Software Development* [4], pubblicato nel 2001, si articola in dodici punti. In esso vengono descritte le metodologie di carattere *Agile*, in contrapposizione a quelle a cascata o altri processi software tradizionali, proponendo un approccio meno strutturato e focalizzato sull'obiettivo di consegnare al cliente, in tempi brevi e frequentemente, software funzionante e di qualità.

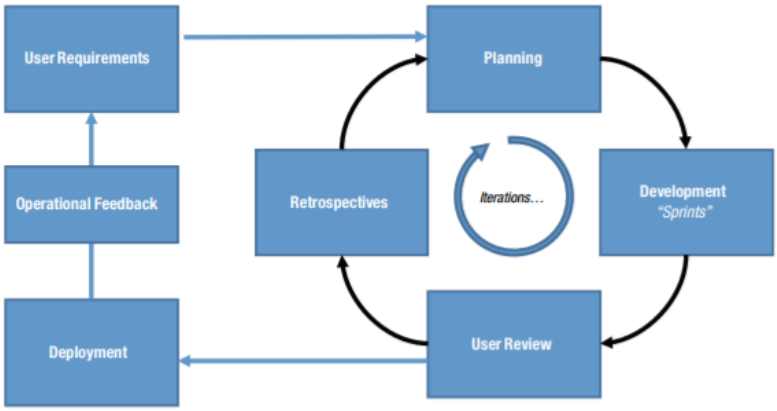

Figura 1.5. Modello di sviluppo Agile (Ravichandran et al., 2016)

Il modello *Agile* è caratterizzato da un pattern iterativo mirato al problem-solving, con feedback continuo sul processo di sviluppo; ciò permette di adattarsi a complessi scenari di business. Ecco alcuni vantaggi del modello:

- Strutturare il processo di sviluppo in micro-unità (*small-batches approach*) consente di rispondere più velocemente a nuovi flussi di informazione, aumentando la frequenza del feedback e di conseguenza la qualità del risultato
- Gli errori o false ipotesi vengono rilevati in anticipo durante il ciclo di sviluppo, diminuendo i costi di correzione
- Il feedback immediato permette ai vari team coinvolti nello sviluppo di migliorare la qualità del software
- Parallelizzando sviluppo e testing, i team *Agile* sono in grado di ridurre i tempi di sviluppo e consegna del software

La flessibilità di questo tipo di metodologia risiede nel fatto che i vari team coinvolti nello sviluppo sono indipendenti tra di loro e sono in grado di intraprendere decisioni proprie sul design architetturale o su come gestire il software in ambienti simili a quelli di produzione. Vengono, dunque, eliminati i bottleneck associati alla propedeucità di esecuzione di alcuni stage rispetto ad altri. Con il modello *Agile*, il rapido supporto all'innovazione si riflette anche in modifiche all'architettura delle applicazioni, al fine di migliorare la scalabilità del software e accelerare la fase di distribuzione. Il design monolitico delle applicazioni tradizionali, in cui tutte le funzionalità sono racchiuse in un unico processo, viene sempre più sostituito da un design distribuito, in cui ogni elemento funzionale è offerto da un servizio differente

(Figura 1.6). L'utilizzo dei micro-servizi consente, dunque, di avere maggiore scalabilità, di avere un meccanismo di fault tolerance robusto e un grado di libertà aggiuntivo per quanto riguarda la scrittura di codice, non essendo più vincolati ad una particolare tecnologia di sviluppo.

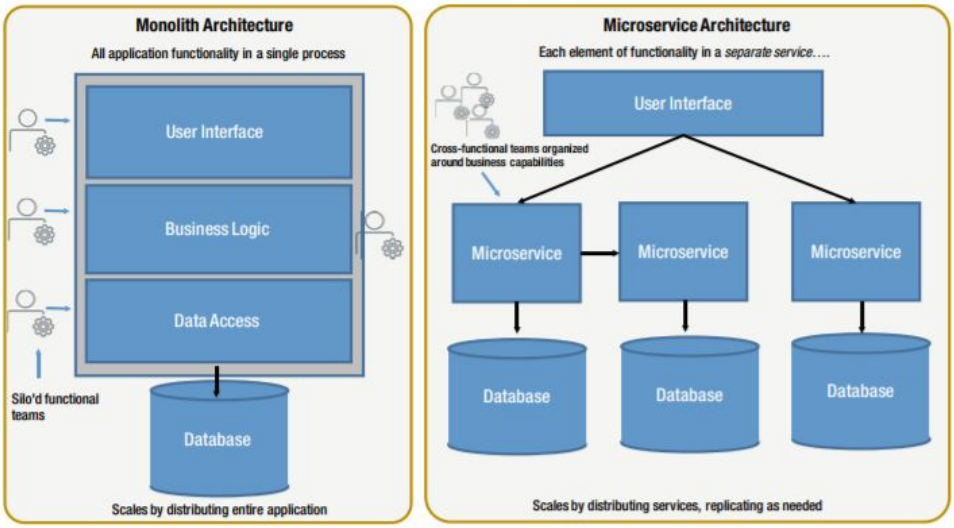

Figura 1.6. Design architetturali a confronto (Ravichandran et al., 2016)

La natura semplice ed elegante dell'architettura a micro-servizi, comporta dunque un certo vantaggio. Tuttavia, ciò aumenta il grado di complessità per ciò che concerne gli aspetti di gestione e collaborazione tra chi sviluppa il servizio – *development* team – e chi si occupa delle fasi di produzione e fornitura del servizio stesso – *operations* team. Sono necessarie importanti considerazioni circa:

- Il supporto costante da parte dei team di sviluppo. In particolare, quando gli altri team si trovano a gestire centinaia di micro-servizi sviluppati in linguaggi differenti, oppure quando hanno necessità, ad esempio, di accedere a nuovi data store o infrastrutture cloud

- Il monitoraggio delle operazioni. Con una molteplicità di micro-servizi, bisogna garantire che molti più processi rimangano performanti

- Il coordinamento delle operazioni. Fornire centinaia di micro-servizi richiede un processo articolato di consegna degli stessi, il quale risulta sempre più difficile da gestire manualmente

Sebbene, dunque, negli ultimi anni sia stato notevole l'avanzamento nell'adottare nuove metodologie di sviluppo del software, come i nuovi strumenti e nuovi design di progettazione *Agile*, gli attriti che emergono tra i vari gruppi IT, coinvolti nel *lifecycle* di una applicazione, si riflettono praticamente in tempi maggiori di consegna, produttività ridotta e sistemi poco funzionali. L'obiettivo non deve e non può essere solo quello di ridurre i lunghi cicli di sviluppo del software attraverso nuove procedure, ma deve essere anche quello di ottimizzare la produttività di ciascun componente appartenente alla catena di rilascio del servizio. Ecco, dunque, che il ***DevOps***, ponendo il proprio focus sia sull'automatizzazione del processo di *deployment*, riducendo i tempi lunghi di produzione del software, sia sulla gestione organizzativa di una infrastruttura IT, massimizzando la collaborazione tra i vari team, sta assumendo un ruolo imperativo per il business attuale. Prima di entrare nel merito della cultura del *DevOps*, è opportuno approfondire gli *anti-pattern* più comuni, nell'ambito di rilascio del software, che ne hanno comportato l'ingente diffusione.

1.4 *Software release*: *Anti-pattern* e Soluzioni

Garantire l'affidabilità di un prodotto software richiede di avere chiare le scelte da evitare in fase di progettazione. La maggior parte delle compagnie, ad esempio, rilasciano software seguendo delle procedure non completamente automatizzate bensì manuali. Può accadere dunque che l'essere umano, per natura meno ripetibile di un algoritmo, non effettui sempre e in maniera sistematica tutti i test necessari in fase di validazione del software, ad esempio. Può accadere, inoltre, che la documentazione manuale risulti poco dettagliata e trasparente, riducendo il livello di collaborazione tra i vari team coinvolti nello sviluppo. Ecco che l'utilizzo di task automatizzati incrementa sia l'efficienza del *deployment* di una applicazione software, sia facilita la comunicazione tra gli attori coinvolti, essendo tutto esplicito in script aggiornati in tempo reale al termine di ogni esecuzione.

Si supponga, per il momento, che tutti i test siano stati effettuati localmente e con successo dagli sviluppatori sulle loro workstation. A questo punto, il software viene posto per la prima volta all'attenzione del team *operations*, il quale si occuperà della configurazone dell'ambiente di produzione e del rilascio. Il rischio che si corre, e che bisogna evitare, è che il software si scopra non funzionante non appena comincia ad essere utilizzato in produzione. Ciò può essere dovuto a configurazioni non corrette dell'ambiente di produzione, oppure ad elementi esterni che si interfacciano con il sistema in modalità non compatibili con quelle definite in fase di sviluppo. Ciò è dovuto, molto spesso, all'assenza di interazione tra chi sviluppa codice e chi deve renderlo funzionale nell'ambiente reale di esecuzione. Una mancanza, dunque, sia

in termini comunicativi sia in termini di gestione delle attività. Pertanto, si rende necessario un sistema di *version control* automatizzato, il quale tracci qualunque tipo di modifica effettuata e che consenta di poter fornire feedback immediati sullo stato del lavoro.

Alla luce di questi accorgimenti, per poter ottenere del software di qualità in maniera rapida ed efficiente, bisogna rendere le *release* software:

- **Frequenti** - Se frequenti, il delta tra *release* differenti sarà minore. Ciò ridurrà il rischio di avere dei rollback costosi in termini sia computazionali che temporali. *Release* frequenti, inoltre, impattano anche sulla frequenza di ricezione del feedback. Come vedremo, ad ogni modifica apportata all'applicazione o ad una configurazione dell'ambiente esistente, seguirà l'esecuzione di un processo di *deployment* finalizzato al rilascio di una nuova versione
- **Automatizzate** - Se il processo di *build*, *deploy*, *test* e *release* non fosse automatizzato, ogni volta il risultato sarebbe differente a causa di configurazioni adoperate in maniera differente

È chiaro, dunque, che il rilascio di un prodotto software debba seguire regole ben definite e meno aleatorie possibili. Come già accennato in precedenza, anche il feedback è una componente essenziale per *release* automatizzate e frequenti. È opportuno, dunque, che vengano rispettati tre importanti criteri:

- Ogni cambiamento, di qualunque tipo, deve generare un trigger per un processo di feedback
- Il feedback deve scattare in tempo reale
- Il team responsabile della consegna del software – *software delivery* – deve ricevere il feedback ed agire immediatamente in sua risposta

Quanto detto finora, riguardo l'automatizzazione del processo di *release* e l'importanza di feedback immediato per ridurre al minimo le tempistiche di fixing dei bug software, sono solo alcuni dei princìpi su cui si fondano la filosofia *DevOps* e la *Continuous Delivery* del software.

1.5 L'etica *DevOps*

Proviamo ad immaginare un mondo in cui i team *Development*, *IT Operations*, *Information Security* e altri collaborino insieme al fine di garantire il successo dell'intera organizzazione. Condividendo tutti lo stesso obiettivo, si potrebbe giungere

a centinaia o addirittura migliaia di distribuzioni giornaliere, a parità di affidabilità e sicurezza.

Questo scenario ideale è ben lontano, purtroppo, da quello del mondo in cui viviamo. Molto spesso i vari team operano in una atmosfera tesa, in cui la mancanza di dialogo e collaborazione impattano negativamente sulla soddisfazione del cliente e sul business in generale. Accade, ad esempio, che chi si occupa dei test o della sicurezza viene coinvolto solamente a giochi fatti, quando è già troppo tardi per risolvere problemi ed errori facilmente evitabili. Il risultato consiste in tempi di stallo e riavvio del processo, in frustrazione e sconforto da parte di chi cerca invano risultati soddisfacenti. Questo è solo un esempio di scenario reale che è possibile trovare in una comune organizzazione IT. La soluzione? Cambiare completamente o migliorare il metodo di lavoro. Ed è proprio in questa direzione che opera il *DevOps*.

Nelle ultime quattro decadi, i tempi e i costi relativi allo sviluppo e alla distribuzione dei servizi sono diminuiti drasticamente [5] (Figura 1.7). Tra il 1970 e il 1980, si impiegavano anni prima di rilasciare un prodotto software. A partire dagli 2000, grazie alle nuove tecnologie e all'adozione dei principi *Agile*, le tempistiche di sviluppo e *deployment* si sono ridotte a mesi o addirittura settimane. Infine, a partire dal 2010, con l'avvento del *DevOps*, è stato possibile utilizzare il software in produzione in poche ore o addirittura minuti, massimizzando l'efficienza e minimizzando il rischio di ottenere un funzionamento indesiderato. Le compagnie che non adottano il *DevOps*, non sono in grado di effettuare centinaia o migliaia di *deployment* giornalieri ma necessitano di mesi o quadrimestri. Questo è dovuto principalmente alla loro incapacità di risolvere il conflitto cronico *Development vs Operations* all'interno della loro organizzazione. Ciò comporta un forte rialzo del cosiddetto ***technical debt***. Esso indica come decisioni sbagliate comportino problemi sempre più difficili da rimediare nel tempo, diminuendo il numero di soluzioni disponibili.

Il fattore critico che contribuisce ad aumentare il *technical debt* è sicuramente quello degli obiettivi discordanti dei team di sviluppo e di quelli che si occupano della configurazione ed effettiva messa in produzione del software. I primi hanno come priorità quella di rispondere rapidamente al cambiamento dei requisiti in risposta alle esigenze di mercato, adottando continuamente modifiche in fase di sviluppo. I secondi, d'altro canto, hanno come priorità quella di fornire un servizio stabile, affidabile e sicuro al cliente, a volte diminuendo drasticamente la flessibilità della soluzione. Come risultato, il ciclo di *delivery* del software avanza sempre più lentamente e sempre meno progetti vengono intrapresi, perdendo terreno nei confronti del mercato che si evolve costantemente. Inoltre, esistono risvolti negativi anche dal punto di vista psicologico. Le problematiche discusse in precedenza, infatti, si traducono in più ore lavorative e nervosismo anche al di fuori dell'ambiente lavorativo. Il *DevOps* si

	1970s–1980s	1990s	2000s–Present
Era	Mainframes	Client/Server	Commoditization and Cloud
Representative technology of era	COBOL, DB2 on MVS, etc.	C++, Oracle, Solaris, etc.	Java, MySQL, Red Hat, Ruby on Rails, PHP, etc.
Cycle time	1–5 years	3–12 months	2–12 weeks
Cost	$1M–$100M	$100k–$10M	$10k–$1M
At risk	The whole company	A product line or division	A product feature
Cost of failure	Bankruptcy, sell the company, massive layoffs	Revenue miss, CIO's job	Negligible

Figura 1.7. Andamento temporale di costi e tempistiche necessarie alla consegna del software

propone, dunque, di migliorare gli aspetti organizzativi e le performance ottenute da ciascuna risorsa, guardando anche al suo benessere personale. Idealmente, piccoli team di sviluppatori implementano in maniera indipendente le proprie soluzioni, ne validano la correttezza in ambienti *production-like*, e demandano il proprio codice alla fase di *deployment* in produzione. Grazie a feedback ottenuti in tempo reale, inoltre, ognuno è in grado di vedere l'effetto delle proprie azioni. In seguito a qualunque azione o modifica effettuata, si pensi ad un semplice *commit* su un *version control system*, una serie di test automatici vengono immediatamente innescati ed eseguiti nell'ambiente di produzione riprodotto. In questo modo, si ha sempre la certezza che il codice e le configurazioni utilizzate siano in uno stato affidabile e compatibile al *deployment*. Il testing automatizzato aiuta chi sviluppa il software a scoprire rapidamente gli errori, solitamente entro qualche minuto, consentendo di risolverli all'istante. Così facendo, il *technical debt* viene ridotto, ogni problema viene risolto appena scoperto e l'obiettivo globale dell'organizzazione assume priorità rispetto a quello dei singoli.

Ulteriore aspetto, da sottolineare in questo scenario, è quello dell'*incremental delivery*. Piccole modifiche vengono effettuate frequentemente, rendendole immediatamente visibili al destinatario del servizio. In questo modo, se qualcosa dovesse andare storto, sarebbe semplice ed immediato effettuare un *rollback* ad una versione

precedente, le cui configurazioni e funzioni sono tracciate nel *version control system* di cui parlavamo in precedenza. Come risultato, la distribuzione del software è facilmente controllabile, reversibile e richiede minore stress. Tuttavia, non è solamente la fase di distribuzione a godere di questi vantaggi. La scoperta e immediata risoluzione dei vari errori, infatti, consente ad ognuno di apprendere come evitarli nel futuro prossimo; una sorta di *training* continuo finalizzato a velocizzare rilasci futuri. Così facendo, si cerca di procedere sempre meno per tentativi, costruendo un metodo scientifico *hypothesis-driven* robusto che tiene conto degli errori da non ripetere.

Al fine di raggiungere gli obiettivi prefissati, c'è bisogno anche di strutturare l'organizzazione dei team in maniera funzionale, garantendo maggiore collaborazione e comunicazione. Anziché formare nuovi team ad ogni *release*, ad esempio, si può pensare di strutturarli a lungo termine, in maniera tale che possano procedere con un approccio consolisato allo sviluppo e al miglioramento continuo del servizio. Invece di una cultura fondata sull'insicurezza, si delinea così una struttura altamente affidabile e collaborativa.

1.6 *DevOps*: il valore di business

I dati pubblicati negli ultimi cinque anni nello *State Of DevOps Report 2016* [6], rivelano l'impatto significativo del *DevOps* sugli aspetti organizzativi e sulle performance di una infrastruttura IT. Nel 2016 il sondaggio ha interessato venticinquemila compagnie in tutto il mondo ed evidenziato i seguenti punti chiave:

- Le organizzazioni che hanno adottato il *DevOps* superano di gran lunga i loro competitor sul mercato, in termini di performance. Esse sono in grado di effettuare, circa duecento volte più frequentemente, nuovi *deployment*. Il *lead time* – tempo che intercorre tra l'inizio del processo e il termine della sua esecuzione – migliora di circa duemila volte mentre la probabilità di fallimento, in seguito all'attuazione di modifiche, si riduce di tre volte (Figura 1.8)

- Alle alte prestazioni sul mercato corrisponde una maggiore *loyalty* degli addetti ai lavori, come dimostrato dall'*employee Net Promoter Score* (eNPS)

- Forte crescita sul mercato, di circa il 50 percento in tre anni. La risposta immediata alle esigenze del mercato, consente di avere ritorni di investimento (ROI – *Return Of Investment*) consistenti. Usando delle metriche opportune è possibile quantificare guadagni e risparmi, come verrà discusso nel paragrafo 1.6.1

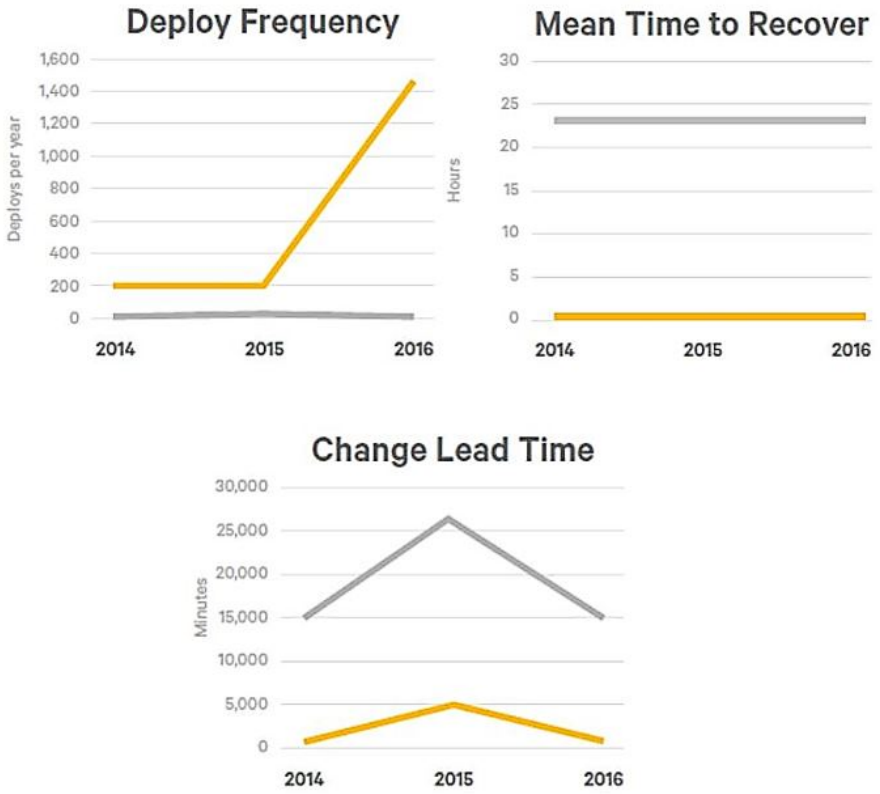

Figura 1.8. Miglioramento delle performance (Puppet Labs, 2016)

- Si risparmia tempo su progetti e revisioni non pianificati, riuscendo ad investire una maggiore percentuale di tempo sulle nuove soluzioni richieste dal mercato (Figura 1.9)

Nel 2016, lo *State of DevOps Report* ha sottolineato come la fase di *development* sia di gran lunga più produttiva se svolta da sviluppatori organizzati in team di piccole dimensioni. Randy Shoup - *director engineer* di Google - ha osservato che, nelle grandi compagnie che adottano *DevOps*, ci sono migliaia di sviluppatori. Tuttavia, essi vengono organizzati e suddivisi in team di piccole dimensioni, estremamente produttivi quasi come se ciascuno di essi fosse una start-up! Nel report viene esaminato non solo il numero di *deployment* giornalieri, bensì anche il numero di *deployment* giornalieri per sviluppatore (Figura 1.10). Le organizzazioni più performanti riescono nell'obiettivo di scalare e aumentare la produttività all'aumentare del numero di sviluppatori; questo è ciò che hanno fatto grandi organizzazioni quali Google, Amazon e Netflix negli ultimi anni. In organizzazioni meno performanti, invece, e meno *DevOps-oriented*, l'effetto è praticamente contrario.

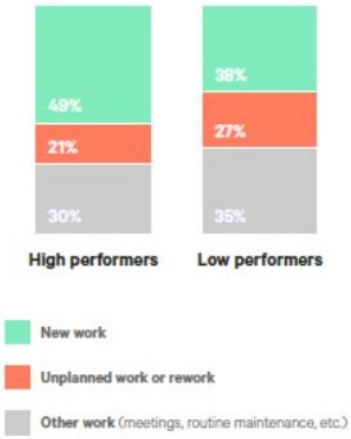

Figura 1.9. Il forte investimento verso le nuove opportunità (Puppet Labs, 2016)

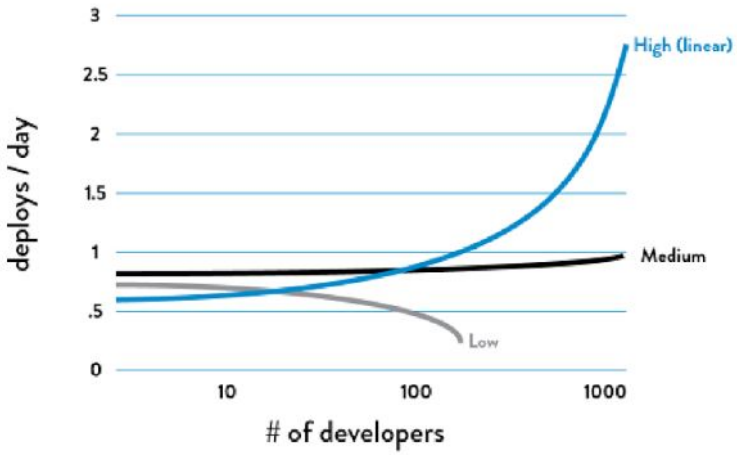

Figura 1.10. Deployment/giorno vs Numero di sviluppatori

1.6.1 Il *Return of Investment* del *DevOps*

Per sopravvivere alle nuove regole di business relative alla produzione di software, il *continuous improvement* e la *continuous delivery* del software si delineano come aspetti fondamentali. Evitare una trasformazione delle logiche di produzione in questo senso, implica venir meno a quelle che sono le richieste di clienti e consumatori.

Ciascun leader aziendale, però, vorrebbe sapere se, investire in una trasformazione tecnologica e organizzativa consistente come il *DevOps*, comporti un ritorno positivo

in termini economici. Utilizzando delle metriche specifiche, è possibile effettuare dei calcoli sui potenziali guadagni delle compagnie che scelgono di adottare tali pratiche. Quello che ci si chiede, fondamentalmente, è se alla fine dei conti il gioco valga la candela o meno. È possibile discutere, inoltre, come il tempo e i costi risparmiati da un *lifecycle* più rapido ed efficiente, vengano investiti nell'ottimizzazione continua della soluzione e delle sue performance.

Il calcolo del ROI si basa, innanzitutto, sui guadagni attesi. Solitamente, questi vengono espressi in termini economici ma è bene tener presente che risparmiare tempo, in questo contesto, è già una sorta di ritorno immediato. Allo stesso modo, aver compreso il valore dell'innovazione è difficile quantificarlo in banconote, ma è sicuramente uno dei guadagni più importanti. L'analista Michael Cote ha affermato che, tentare di definire quantitativamente il ROI del *DevOps*, sia simultaneamente 'assurdo e fondamentale'. Troppe variabili, infatti, sono coinvolte nelle trasformazioni introdotte con il *DevOps* e sarebbe impreciso quanto impossibile ridurne l'impatto ad un valore economico. Lo stesso Cote, in una delle sue guide, definisce il *DevOps* un processo non misurabile in termini di ritorno dell'investimento. Il suo valore aggiunto, dunque, non è stimabile con precisione.

Tuttavia, oltre al successo dei progetti intrapresi e conclusi adottando la nuova etica, un indicatore numerico importante in fase di valutazione circa l'utilizzo del *DevOps*, può essere il cosiddetto MTTR – *mean time to repair* (Figura 1.11). Il critico Cristopher Null, nel suo articolo *How to Measure DevOps ROI* [7], lo definisce come un indicatore del successo di questa nuova rivoluzione in ambito *software-development*. Il tempo impiegato a risolvere gli errori è, infatti, proporzionale al tempo di inattività e di stallo del ciclo di *deployment*, il quale diminuisce il numero di *deployment* effettuati nel periodo stabilito (Figura 1.11).

L'espressione 'Il tempo è denaro' calza a pennello in questo contesto, e riesce a dare l'idea di quanto il ROI sia fortemente condizionato dal fattore temporale.

1.7 Conclusione

Le metodologie di sviluppo del software hanno subito notevoli evoluzioni nel corso degli anni, avendo l'obbligo di adattarsi ad un business sempre più volto alla digitalizzazione di servizi e prodotti software. Modificare il modello del ciclo di sviluppo del software, però, non basta. Si rende necessaria una radicale trasformazione organizzativa che risulti il più efficace possibile nel rilasciare software di qualità in tempi minimi. L'etica *DevOps* sta assumendo sempre più il ruolo di requisito di una mentalità vincente all'interno del mercato. Il principio su cui si fonda, quello

	High Performers	Medium Performers	Low Performers
Deployment frequency	1,460 deploys yearly	32 deploys yearly	7 deploys yearly
Change failure rate	7.5%	38%	23.5%
Mean time to recover	1 hour	24 hours	24 hours*
Outage cost	$500,000/hr	$500,000/hr	$500,000/hr
Total cost of downtime per year	$54.75M	$145.92M	$19.74M
Downtime cost per deployment	**$37.5K**	**$4.56M**	**$2.82M**

Figura 1.11. Costi dell'inattività correlata agli errori nel lifecycle di produzione (Puppet Labs, 2016)

della *continuous delivery* di soluzioni software, è l'elemento chiave per puntare nella giusta direzione.

Capitolo 2

Struttura e configurazione di una *Continuous Delivery Pipeline*

2.1 Introduzione

In questo capitolo, discuteremo le tecnologie e i dettagli implementativi a supporto del *workflow* '*Development into Operations*', riducendo i rischi associati al *deployment* e alla consegna del software in produzione. La *Continuous Delivery* include le basi per l'implementazione della pipeline automatizzata di rilascio del software, assicurando l'esecuzione automatizzata dei test che validano costantemente lo stato del software e consentendo agli sviluppatori di integrare continuamente - *continuous integration* - il loro codice quotidianamente.

Dopo aver illustrato l'anatomia di una *deployment pipeline*, analizzeremo nello specifico ciascuno stage che la compone: dalla *continuous integration* all'*automated testing*, progettando e automatizzando le procedure per *release* a basso rischio.

2.2 L'Architettura della *Deployment Pipeline*

La *Deployment Pipeline* può essere definita come la modellizzazione astratta di quel processo che il software attraversa, partendo da un sistema di controllo di versione sino ad arrivare nelle mani dell'utente finale. Ogni modifica al software deve attraversare vari stage, affinchè esso possa essere rilasciato nella sua versione aggiornatae validata.

La Figura 2.1 mostra un esempio di pipeline che cattura l'essenza della *continuous delivery* del software. [8]

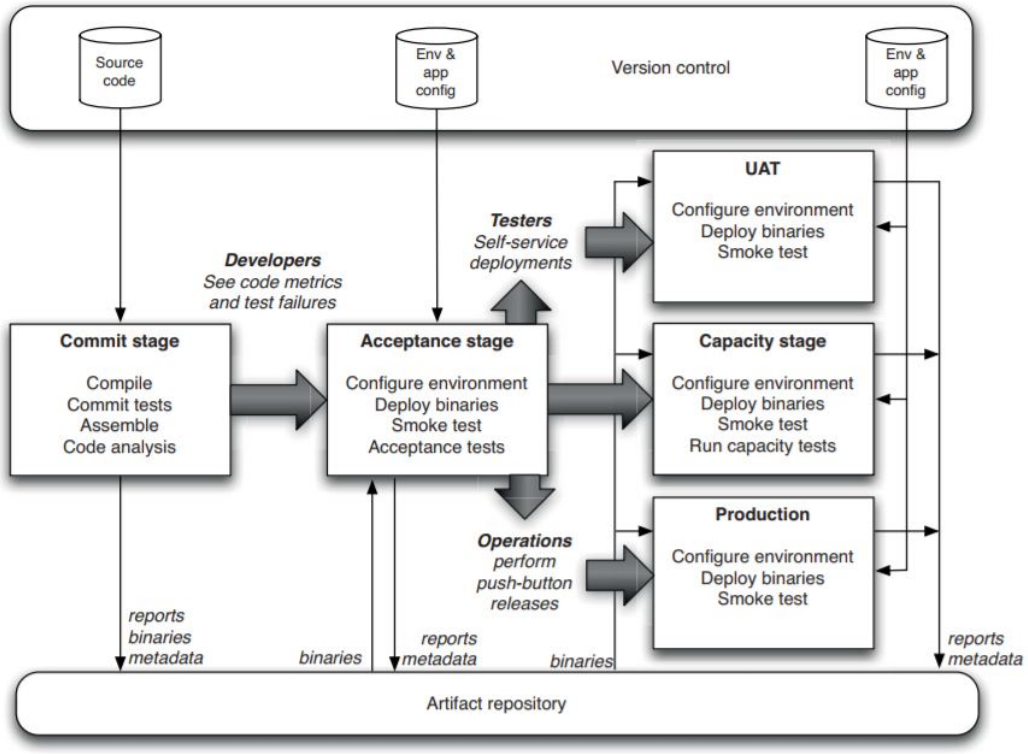

Figura 2.1. Flusso di esecuzione in una pipeline di rilascio del software (Humble et Farley, 2011)

Il processo ha inizio non appena uno degli sviluppatori esegue il *commit* di una modifica nel sistema di *version control*. Il sistema di *continuous integration*, a questo punto, viene innescato e una nuova istanza della pipeline deve essere eseguita. Ogni build, in seguito a qualunque tipo di modifica del codice, deve attraversare una sequenza di test di diversa natura, con l'obiettivo di capire se possa essere rilasciata o meno (Figura 2.2).

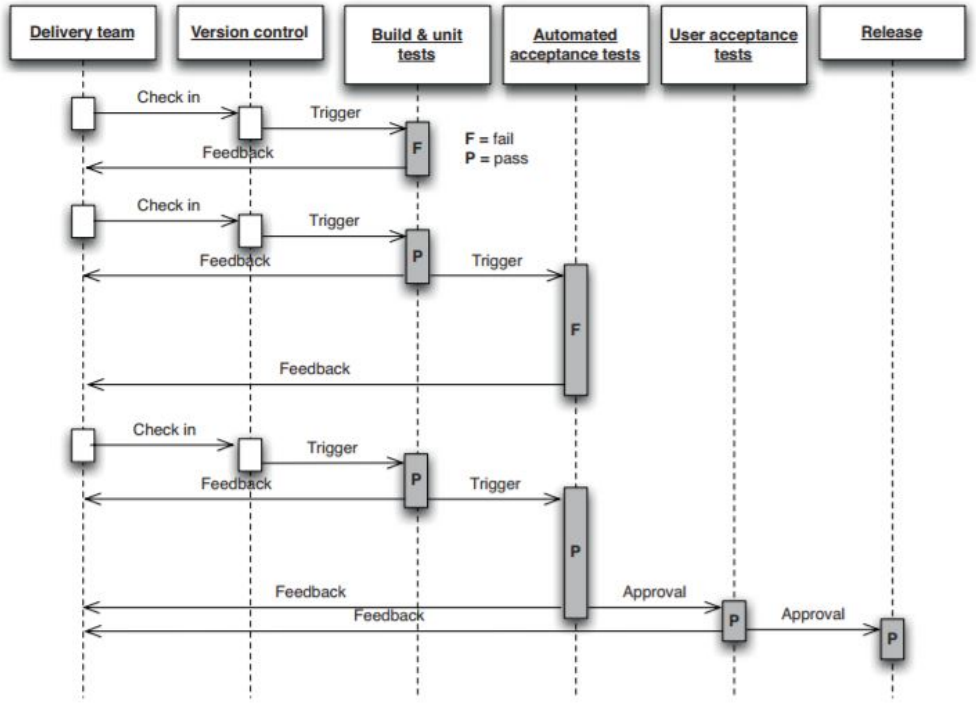

Figura 2.2. Propagazione delle modifiche all'interno della pipeline (Humble et Farley, 2011)

Lo scheduling intelligente alla base del sistema di *continuous integration*, consente di validare il codice e aggiornare il *version control system*, qualora tutti i test venissero superati; in caso contrario, il team di sviluppo verrà notificato istantaneamente del fallimento. Una volta che il problema verrà risolto, la sequenza verrà reiterata. L'efficienza dello scheduling, inoltre, è ancor più evidente in scenari come quelli in Figura 2.3, in cui grossi team di sviluppatori eseguono *commit* frequentemente e vicini nel tempo tra loro. [8]

Prima di entrare in merito a ciascuna fase della pipeline in questione, è opportuno definire e descrivere in dettaglio i tre pilastri [9] alla base di qualsiasi soluzione di *release automation*:

- *Configuration Management* - Utilizzo di un *version control system*
- *Testing Strategy* - Implementazione di una adeguata struttura di testing, a vari livelli
- *Continuous Integration* - Processo automatizzato per la compilazione e validazione del codice

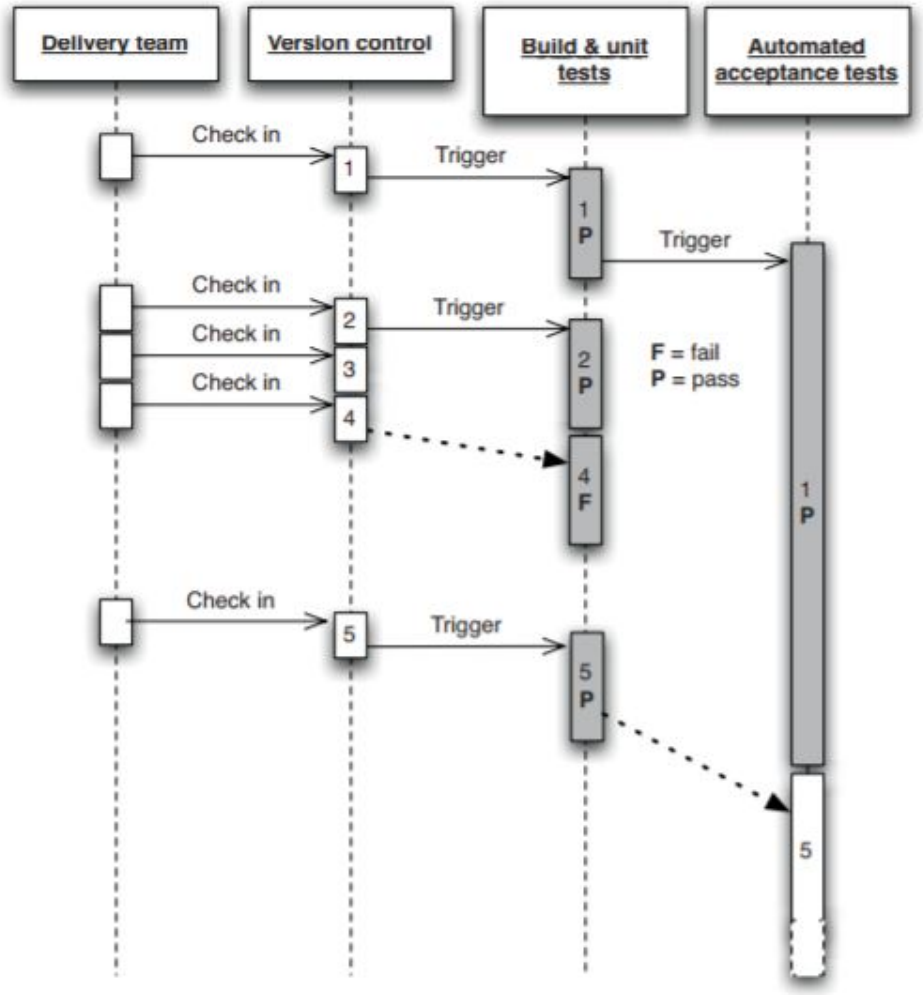

Figura 2.3. Efficienza dello scheduling in un sistema di *continuous integration* (Humble et Farley, 2011)

2.2.1 *Configuration Management* e controllo di versione

Molto spesso si utilizza il termine *source code management* come sinonimo di *configuration management*. Tuttavia, un sistema di *configuration management* oppure di *version control* è qualcosa di più complesso. Esso, infatti, permette di memorizzare non solo il codice sorgente (*source code*) di una applicazione, bensì tutte le informazioni che servono a ricreare ambienti *production-like* in qualunque stage della pipeline ci troviamo. Questi ambienti devono essere ricreati in maniera automatizzata, idealmente on demand a partire da script e informazioni di configurazione memorizzate all'interno di un sistema di *versioning*, senza richiedere alcun intervento umano. In un contesto come quello della *release automation*, dunque, non si può fare a meno

di un sistema come quello di *version control*. Tutto ciò che verrà discusso successivamente, dalla *continuous integration* all'*automated testing*, dipenderà dall'avere memorizzato in un repository tutto ciò che è correlato al ciclo di vita di un progetto.

Molto spesso accade che gli aspetti più catastrofici di una *release* software emergano nell'effettivo ambiente[2] di produzione, quando l'applicazione è sottoposta a un workload realistico. Quello che i team di sviluppo preferirebbero, è avere a disposizione degli ambienti *production-like* per il testing, ancor prima che il software venga rilasciato effettivamente in produzione. Così facendo, essi sarebbero in grado di eseguire e testare continuamente il loro codice, riproducendo l'ambiente di produzione sulle loro workstation e ricevendo feedback immediato sulla qualità del loro lavoro. In assenza di un sistema di *versioning*, gli sviluppatori dovranno attendere che i sistemisti, appartenenti al team *Operations*, foniscano le configurazioni necessarie alla creazione di un ambiente di testing *production-like*. Può accadere, purtroppo, che gli sviluppatori ricevano queste informazioni troppo tardi - si parla addirittura di settimane - non riuscendo a condurre test adeguati, oppure ottenendo configurazioni scorrette che incrementano il *lead time* discusso nel capitolo precedente. Con l'utilizzo di un sistema per il controllo versione, invece, diventa possibile riprodurre ambienti di sviluppo, testing e produzione in tempo reale - si parla di minuti - senza dover introdurre alcun tipo di bottleneck.

Anzichè attendere che gli ambienti vengano configurati manualmente prima di poterli riprodurre, è possibile utilizzare una procedura automatizzata che consente di:

- Creare *on demand* una network di macchine virtuali, ad esempio utilizzando una immagine VMware o eseguendo uno script Vagrant
- Utilizzare tool per *infrastructures as code*, quali Ansible, Chef o Puppet
- Riprodurre on demand ambienti di sviluppo, testing, produzione, partendo da immagini di macchine virtuali o container (es. Vagrant, Docker)
- Eseguire servizi sul cloud (es. Amazon Web Services, Google App Engine, Microsoft Azure)

L'obiettivo, inoltre, non è solo quello di riprodurre ambienti in maniera rapida, bensì anche quello di facilitare la collaborazione e la comunicazione tra i vari team, affinchè software affidabile, stabile e sicuro venga rilasciato. Il primo *version control*

[2]In questo contesto, per ambiente si intende tutto ciò che permette all'applicazione di essere eseguita: database, immagini di sistemi operativi, configurazioni di rete, virtualizzazione e altri file di configurazione utilizzati

system è stato SCCS (*Source Code Control System*), un tool proprietario di UNIX risalente al 1970. SCCS è stato sostituito da RCS (*Revision Control System*) prima, e da CVS (*Concurrent Version System)* poi. Questi tre tool sopravvivono ancora oggi ma sono utilizzati pochissimo. Sono altri, infatti, quelli maggiormente utilizzati. Si tratta di sistemi sia open source che proprietari, compatibili con qualunque piattaforma: Subversion, Mercurial e Git ne sono un diffuso esempio. Ciascuno di essi memorizza i vari *commit* o *revisions* effettuati su codice, asset, o altri tipi di documenti utilizzati nello sviluppo del software. Ogni *revision* contiene informazioni su chi ha effettuato la modifica, quando è stata effettuata e cosa è stato modificato; ciò consente, ad esempio, di poter confrontare facilmente diverse versioni del codice oppure ripristinare una versione precedente. Un sistema di *versioning*, dunque, fotografa in maniera precisa lo stato attuale di un sistema, disponibile a ciascun team coinvolto nella produzione del software. In dettaglio, al suo interno, devono essere memorizzati:

- Codice e dipendenze (es. librerie, statiche e dinamiche)
- Tutti gli script utilizzati per la creazione e modifica di database
- Tutti i tool, e le relative configurazioni, utilizzati per la creazione di ambienti virtuali
- Tutti i file utilizzati per la creazione di container (es. Dockerfile)
- Tutti gli script utilizzati per le varie tipologie di testing, manuali e automatizzati
- Tutti gli script utilzzati per il *packaging* e il *deployment* di un'applicazione
- Qualunque tipologia di *artifact* e metadato (es. documentazione dei requisiti, procedure di *deployment*, file di log, note di *release*)

Tutto questo ci consente di risolvere una serie di problematiche in fase di preproduzione, riducendo il rischio di doverle risolvere quando si è già in una fase avanzata del processo di *deployment*. Il corretto utilizzo di un sistema di questo genere, insieme alla definizione di una opportuna strategia di testing, rappresenta un requisito fondamentale per le pratiche di *continuous integration* che verranno descritte più avanti.

2.2.2 Automatizzazione della fase di *testing*

Arrivati a questo punto, chi sviluppa il codice è in grado di eseguirlo e testarlo continuamente in ambienti *production-like*, tracciando qualunque modifica in un sistema di *version control*. Tuttavia, i test eseguiti dai *developers* in fase di sviluppo,

rappresentano una sola delle categorie di testing cui il software viene sottoposto attraversando la pipeline. Solo quando la fase di *development* viene portata a termine, infatti, il team di controllo qualità eseguirà una seconda fase di testing. Quello che spesso accade, però, è che questa seconda fase venga eseguita tipicamente due o tre volte in un anno, data la scarsa frequenza con cui uno sviluppatore rilascia il suo codice. Si rischia, così, che agli sviluppatori stessi vengano notificati gli errori dopo mesi, annullando i vantaggi intrinsechi della *continuous integration*.

Una strategia automatizzata per il testing, invece, mira a risolvere il problema, evitando di sprecare una quantità di tempo proporzionale alle linee di codice che vengono aggiunte in fase di sviluppo. Non appena uno sviluppatore esegue un *commit*, delle suite contenenti migliaia di test vengono eseguite in maniera automatica sul codice modificato. Se tutti i test vengono superati, il codice è già pronto per il *deployment* in produzione. A partire dal 2013, l'utilizzo del testing automatizzato e della *continuous integration* ha permesso, in una grande organizzazione come Google, di coordinare migliaia di piccoli team tra loro, tutti impegnati simultaneamente nello sviluppo, integrazione di codice, testing e *deployment* in produzione. Il loro codice è condiviso in un'unica repository, composta da miliardi di file, in cui il codice viene continuamente compilato e revisionato; circa il 50% di questo codice viene modificato in un solo mese. Per dare un esempio di quanto questo approccio possa essere produttivo, ecco alcune delle loro statistiche:

- 40 mila commit al giorno
- 50 mila build giornaliere
- 120 mila suite utilizzate per il testing *automated* e 75 milioni di test complessivi
- Più di 100 mila ingegneri che progettano nuovi tool per la *continuous integration* e *release automation* del software, al fine di incrementare la produttività di ogni singolo sviluppatore

L'importanza di seguire questo approccio è dunque evidente, visti gli eccellenti risultati. Prima di entrare nei dettagli della *continuous delivery pipeline*, analizzando il flow di esecuzione delle diverse categorie di test, cerchiamo di descriverle singolarmente, dalla più veloce alla più lenta:

- **Unit tests** - Essi vanno a testare un singolo metodo, classe o funzione, notificando lo sviluppatore circa la correttezza del proprio codice. I test unitari sono scritti e gestiti esclusivamente dallo sviluppatore all'interno della fase di *development*. Sono la categoria di test più rapida in quanto non interagiscono con alcun'altra componente, quali database o filesystem, che richieda

delle chiamate esterne. Ciò implica, d'altro canto, un testing poco dettagliato che si limita a verificare localmente la corretta esecuzione del codice. Nel *deployment* di applicazioni moderne, le quali comunicano con diversi database oppure scambiano dati di diversa natura con servizi esterni, i test unitari non sono sufficienti da soli a garantire la totale affidabilità del software

- **Functional acceptance tests** - Essi rispondono alle domande [8] "Come faccio a sapere se ho finito?" degli sviluppatori, e "Ho ottenuto quello che volevo?" dell'utente. A differenza dei test unitari che vanno a validare una singola e isolata parte dell'applicazione, con i test funzionali si dimostra che l'applicazione si comporta come il cliente ha richiesto e non che funziona semplicemente come lo sviluppatore ha programmato

- **Integration tests** - Sono la tipologia di *automated tests* meno utilizzata, data la loro elevata complessità e onerosità computazionale. Molto spesso, infatti, dopo la corretta esecuzione dei test funzionali, si preferisce sottoporre l'applicazione alla fase di testing manuale (es. test esplorativi, test sull'interfaccia grafica). Questo perchè non tutto può essere automatizzato: aspetti come l'usabilità, l'aspetto, la percezione, sono difficili da automatizzare ed è per questo che i test manuali devono continuare ad essere utilizzati, seppure in percentuale decrescente

L'approccio ideale sarebbe quello di eseguire prima i test più veloci, *unit tests*, e poi quelli che richiedono più tempo, rispettivamente *acceptance tests* e *integration tests*. La differenza in termini di velocità di esecuzione tra queste categorie, si riflette anche nella loro capacità di inviare il prima possibile un feedback agli sviluppatori circa gli errori rilevati in fase di testing. Qualora, infatti, lo sviluppatore risolvesse velocemente un bug riscontrato dagli *integration tests* (o analogamente dagli *acceptance tests*), dovrà comunque attendere qualche ora prima di capire se esso è stato risolto o meno. Al contrario, se il bug fosse rilevato dai test unitari, il feedback e la risoluzione degli errori sarebbero pressochè immediati. Sarebbe ideale, dunque, che la maggior parte degli errori venisse rilevata dagli *unit tests* ma molte volte, la piramide di testing, assume la forma inversa (Figura 2.4). Una possibile contromisura, può essere quella di creare un nuovo caso di test unitario ogni volta che un errore viene scoperto dalle tipologie di test più lente; si cerca, in questo modo, di allargare la base della piramide non ideale il più possibile, incrementando la copertura degli test più rapidi attraverso una logica di *caching* degli errori. Al fine, inoltre, di eseguire i test il più velocemente possibile, si potrebbe pensare di eseguirli in parallelo [8] (Figura 2.5), potenzialmente distribuendoli su diversi server.

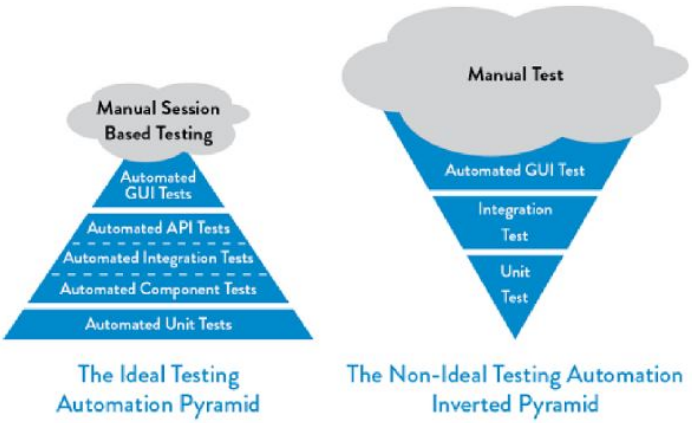

Figura 2.4. La piramide ideale e non ideale dell'*automated testing* (Kim et al., 2016)

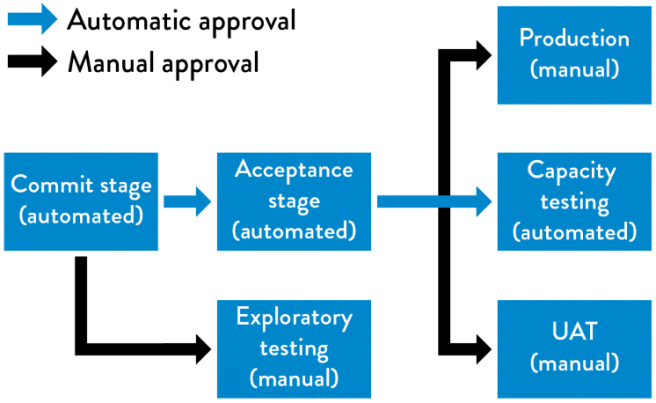

Figura 2.5. Esecuzione in parallelo di manual e automated tests (Kim et al. 2016)

2.2.3 I vantaggi della *Continuous Integration*

Nella sezione precedente, abbiamo discusso le strategie di testing che consentono agli sviluppatori di ottenere feedback continuo sulla qualità del loro lavoro. Il problema da risolvere, e che la *continuous integration* si propone di risolvere, emerge all'aumentare del numero di sviluppatori e del numero di *branches* su cui essi lavorano in maniera parallela.

In un sistema di *version control*, per *branch* si intende una duplicazione del codice che ogni sviluppatore preferisce utilizzare per non interferire, con le proprie modifiche, direttamente sul codice originale. Soltanto dopo che le modifiche vengono portate a termine, il codice di ciascun *branch* viene integrato - *merging* - sul

ramo originale denominato *master* (Figura 2.6). Ciò consente, sicuramente, di parallelizzare lo sviluppo ma può comportare dei problemi in fase di integrazione delle modifiche. Più tempo, infatti, gli sviluppatori isolano le modifiche, più problemi ci saranno in fase di *merging* sul nodo principale.

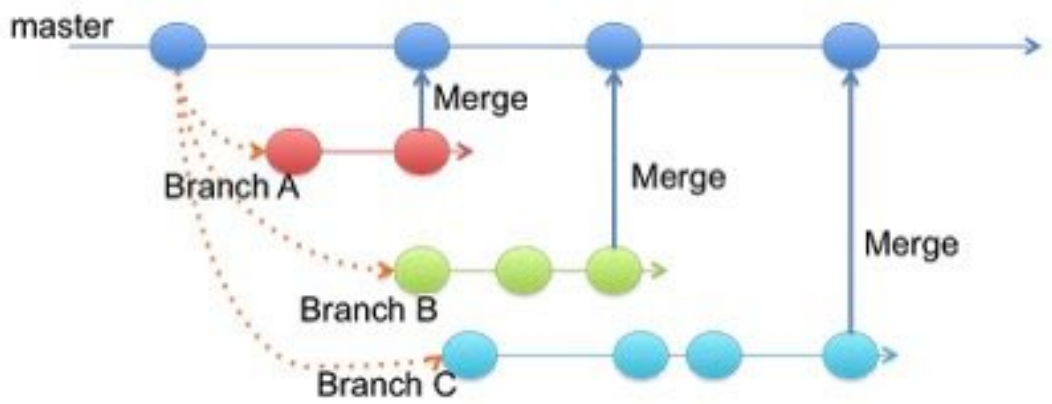

Figura 2.6. *Branching* e *Merging* in un sistema di controllo versione

La *continuous integration* fà in modo, dunque, che il *merging* diventi una pratica quotidiana per gli sviluppatori, e non qualcosa da eseguire il più lontano possibile nel tempo. Proviamo a darne un esempio, analizzando l'esperienza di Gary Gruver [10], ingegnere della divisione HP LaserJet Firmware che si occupa del rilascio di firmware per stampanti, scanner e altre tipologie di device. Il suo team comprende centinaia di sviluppatori tra USA, Brasile e India. Per anni, dato l'elevato numeri di componenti, sono stati inefficienti nel rilasciare nuove soluzioni, rilasciando solamente due firmware all'anno e perdendo terreno nei confronti del mercato. Gruver ha stimato che solamente il 5% del tempo veniva investito nel creare nuove soluzioni, il restante tempo veniva utlizzato per lavori non produttivi associati al *technical debt* discusso nel capitolo precedente, quali:

- 20% del tempo impiegato nel riprogettare l'organizzazione del lavoro poco produttivo
- 25% del tempo impiegato nella gestione di *porting code*[3], distribuito su *branches* separati
- 10% del tempo impiegato nel *merging* dei diversi *branches*
- 15% del tempo impiegato nel testing manuale

[3] «Con il termine *porting code* si intende del codice portabile ossia eseguibile in ambienti con caratteristiche hardware o software differenti da quello in cui il codice è stato sviluppato.» (Wikipedia)

La creazione di un sistema di *continuous integration*, supportato da tutta la serie di *automated tests* discussi in precedenza, ha cambiato radicalmente le cose. Gruver ha ammesso che l'adozione di un modello di sviluppo *trunk based*, in cui viene utilizzato un unico *branch* condiviso da tutti, ha rappresentato una vera e propria rivoluzione. Molti ingegneri del software non credevano che ciò fosse possibile, preferendo l'utilizzo di più *branches* privati.; tuttavia, come riportato nel *2017 State of DevOps Report*, lo sviluppo *trunk-based* comporta un elevato rate di sviluppo, maggiore soddisfazione all'interno del team e rischio minore di malfunzonamenti. In questo scenario, ogni tipo di modifica viene continuamente integrata, eseguita e validata in tempo reale, avviando continuamente nuove istanze della pipeline di rilascio al fine di ottenere qualità elevata in tempi estremamente ridotti. L'intero processo ha consentito di:

- Aumentare il numero di build giornaliere, passando dal rilascio di una sola build a più di venti
- Incrementare il numero di *commit* per giorno per sviluppatore, da circa dieci a più a di un centinaio
- Aumentare il numero di righe di codice aggiunte o modificate dagli sviluppatori ogni giorno, da circa 75 mila a più di 100 mila linee
- Ridurre significativamente i tempi associati ai test di regressione[4], da circa sei settimane ad un solo giorno!

L'esperienza di Gruver dimostra come la *continuous integration*, abbinata al *versioning control* e all'*automated testing*, permetta di raggiungere risultati mai visti prima. Il rate di sviluppo che aumenta del 140% e i costi che si riducono del 40%, dimostrano l'effettiva necessità di un sistema di questo genere, il quale, più che un tool da utilizzare, deve essere una disciplina da adottare.

2.3 Gli stage della pipeline: il *Commit Stage*

Dopo aver presentato i tre attori principali, necessari alla realizzazione di una soluzione per la *release automation*, discutiamo ora i vari stage di una *deployment pipeline* e cerchiamo di capire in quali fasi questi attori vengono coinvolti.

[4]I test di regressione o *regression tests*, abbracciano trasversalmente tutta la categoria degli *automated tests*

Il *commit stage* (Figura 2.7) ha inizio quando il sistema di *continuous integration* rileva una modifica allo stato del progetto, come ad esempio un *push* (equivalente di *commit*) sul sistema di *version control*, e termina con:

- Un report contenente gli errori (in caso di fallimento)
- La creazione di metadati, file binari o *artifacts*[5] in generale da memorizzare in un repository apposito e da utilizzare nelle fasi successive (in caso di successo)

Idealmente, l'esecuzione di questa fase dovrebbe durare meno di cinque minuti, o comunque non più di dieci minuti in casi eccezionali. Il *commit stage* rappresenta, dunque, il punto di ingresso alla pipeline e prevede l'esecuzione di una serie di task, orchestrati dagli script eseguiti per opera del server di *continuous integration*:

- Compilazione, se necessaria, del codice ed esecuzione di *unit tests* sul codice che è stato integrato
- Analisi della qualità del codice, al fine di rilevarne bug e vulnerabilità
- Creazione e salvataggio degli *artifacts*

Se l'obiettivo principale di una *deployment* pipeline è di eliminare quelle build non adatte alla produzione, il *commit stage* si presenta come primo discriminante in questo senso. Gli sviluppatori riescono così a ottenere un primo veloce e fondamentale feedback. Il fallimento dell'esecuzione dei test unitari in questo stage può essere attribuito a diverse cause: un errore di sintassi all'interno del codice, un problema di configurazione dell'ambiente di esecuzione oppure un errore di semantica all'interno dell'applicazione. Indipendentemente dal tipo di problema, gli sviluppatori devono essere notificati il prima possibile al fine di poter correggere gli errori e minimizzare i tempi di stallo della pipeline.

Definiti gli input e le possibili cause di successo o fallimento di questa prima fase, soffermiamoci adesso sugli output del *commit stage*. Metadati, report e quant'altro devono essere condivisi con gli altri team, in maniera da poterli utilizzare nelle fasi successive della pipeline di rilascio. La prima cosa che si potrebbe pensare è quella di effettuare il *push* degli output sul sistema di *version control*, tuttavia questa non è la soluzione ideale in quanto si rischierebbe di generare un loop di inneschi della pipeline. Pertanto, si rende necessario utilizzare un apposito repository per

[5] «In informatica, e in particolare in ingegneria del software, un artefatto è un sottoprodotto che viene realizzato durante lo sviluppo software. Sono artefatti i casi d'uso, i diagrammi delle classi, i modelli UML, il codice sorgente e la documentazione varia, che aiutano a descrivere la funzione, l'architettura e la progettazione del software.» (Wikipedia)

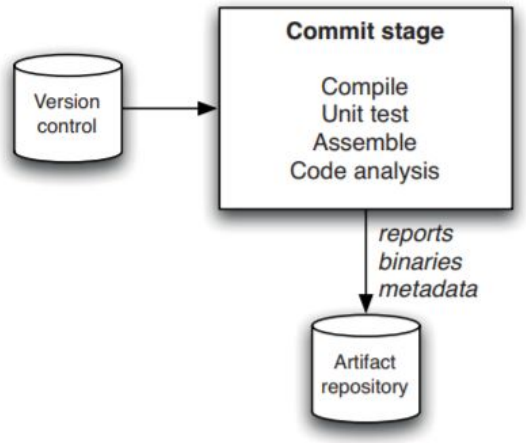

Figura 2.7. Il *commit stage* (Humble et Farley, 2011)

gli *artifacts*. I più moderni server di *continuous integration*, come Jenkins che verrà utilizzato nel capitolo successivo, forniscono un proprio *artifact repository*, di cui è possibile configurare le proprietà: è possibile decidere, infatti, quale tipologia di artifact memorizzare, per quanto tempo e in seguito all'esecuzione di quale evento. In alternativa, se non si vuole utilizzare il server di *continuous integration* a tal scopo, è possibile utilizzare diversi tipi di *repository manager*, quali *Nexus*[6] o altri tool *Maven-style*[7].

2.4 Gli stage della pipeline: l'*Automated Acceptance Test Stage*

Il software che supera con successo la fase di *commit* e i test unitari da essa previsti, viene sottoposto agli *acceptance tests* (Figura 2.8). Per la prima volta, nel corso di esecuzione della pipeline, troviamo una tipologia di test non più *developer-facing* bensì *business-facing*. [8]

Gli *acceptance tests*, di fatto, simulano l'interazione dell'utente finale con il software in produzione, fornendo già un primo feedback sul corretto adempimento al comportamento richiesto. In questa fase, viene riprodotto l'ambiente di utilizzo dell'applicazione, utilizzando le stesse configurazioni dell'ambiente di produzione vero

[6]URL: https://www.sonatype.com/nexus-repository-sonatype
[7]URL: https://maven.apache.org/repository-management.html

e proprio. L'applicazione viene così testata, sottoponendola a diversi casi di test automatizzati, al fine di poterne misurare il grado di usabilità.

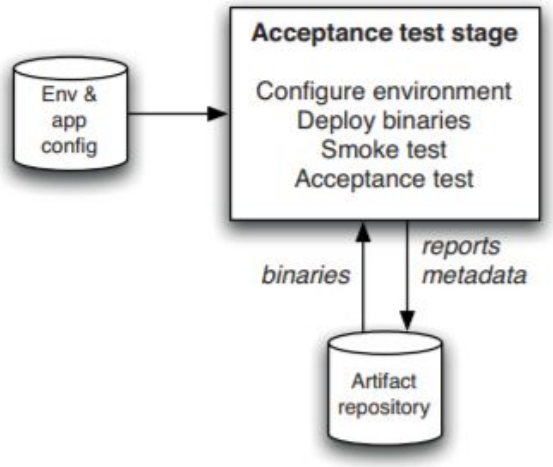

Figura 2.8. L'*acceptance test stage* (Humble et Farley, 2011)

In uno dei suoi articoli, Dan North definisce un nuovo paradigma di sviluppo orientato al testing di un applicazione. Anzichè utilizzare il ben noto paradigma TDD (*Test Driven Development*) per la stesura dei test automatici, il BDD (*Behaviour Driven Development*) definito da North pone l'accento sui criteri di comportamento che un'applicazione deve adottare. Questi cosiddetti *acceptance criteria* devono essere definiti nella forma ***Given -When-Then*** [11]. Nel testing di un'applicazione, con *Given* si intende lo stato dell'applicazione iniziale, prima dell'esecuzione del test. Il *When* simboleggia l'interazione dell'utente con l'applicazione, mentre con il *Then* viene indicato lo stato che l'applicazione dovrà assumere in seguito a quella specifica interazione. Il compito del caso di test, dunque, è quello di verificare che una certa applicazione, inizialmente in uno stato *Given*, assuma il nuovo stato descritto nella clausola *Then* dopo aver eseguito l'azione descritta nel *When*.

Seguendo questo tipo di approccio, i task principali di questa seconda fase della pipeline sono:

- Definire gli *acceptance criteria* nella forma appena discussa. Da sottolineare che la loro definizione è rapportata ai requisiti richiesti dal cliente o utente finale
- Tradurre questi criteri in degli script che possano essere eseguiti in maniera automatica al termine del *commit stage*

2.5 Stage successivi e *release* del software

L'*acceptance test* stage, rappresenta un punto di arrivo parziale della pipeline. Giunto in questa fase e superandola con successo, il software è a tutti gli effetti un candidato idoneo alla *release*, essendo già pronto all'utilizzo da parte dell'utente finale. In molti processi di *release*, però, si preferisce inserire un ulteriore fase di testing manuale. Più che uno stage determinante per il rilascio definitivo, il *manual testing* si presenta come una ulteriore validazione dei test automatici svolti nella fase precedente. Il ruolo del tester umano è quello di verificare, manualmente, il soddisfacimento degli *acceptance criteria* definiti e verificati già in maniera automatica. Una sorta di prova del nove, dunque, in cui il tester può essere visto come l'utente finale che partecipa attivamente al *deployment* del software.

Oltre alla fase di *manual testing*, prima della *release* definitiva, possono essere inseriti altri step ulteriori. Ne è un esempio la validazione di requisiti non funzionali, quali policy di sicurezza o di altro genere, cui il servizio deve essere conforme: si parla in questo caso di *nonfunctional testing*. Se invece si è meno interessati alla *compliance* e si preferisce capire come il sistema reagisce alla variazione del carico di lavoro in produzione, in questo caso si effettuano dei test di capacità (*capacity testing*). E' chiaro dunque che, successivamente al *commit stage* e l'*acceptance test stage*, indispensabili in qualunque processo di *release automation*, aumenta il grado di libertà circa la costruzione della pipeline. Ciò varia in relazione alle esigenze e ai requisiti da soddisfare, dipendentemente dal campo applicativo in cui il software dovrà essere utilizzato.

Giunti a questo punto, siamo in grado di:

- Creare un piano di *release* gestito in collaborazione da ogni risorsa, tra cui sviluppatori, tester, sistemisti e chiunque altro coinvolto nello sviluppo, manutenzione e *delivery* del software
- Minimizzare il livello di errore, automatizzando tutto quello che è possibile all'interno del processo di *deployment*
- Simulare le procedure di un ambiente di produzione, testando configurazioni e build in ambienti *production-like*
- Migrare facilmente il software in produzione nel caso di aggiornamenti, ripristinare vecchie *release* in caso di problemi

Alla luce di quanto detto, l'ultimo stage della pipeline, quello della *release* vera e propria, non è altro che un naturale risultato del corretto funzionamento e della

corretta implementazione degli stage precedenti. Di seguito (Figura 2.9) viene riportata una visione d'insieme della *deployment pipeline*, in cui vengono inseriti i vari attori e i vari passaggi effettuati prima di giungere al risultato finale [8].

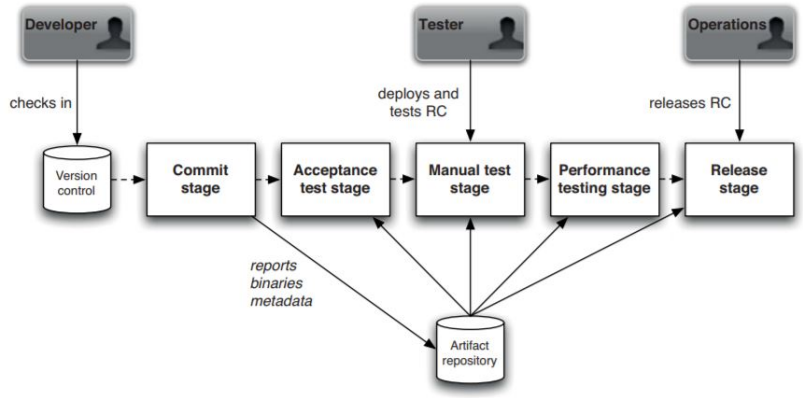

Figura 2.9. Deployment Pipeline: dal *commit* iniziale alla *release* del software (Humble et Farley, 2011)

Riepilogando, in ordine, i vari step:

- Qualcuno tra gli sviluppatori esegue un *commit* sul sistema di *version control*
- Il server di *continuous integration* avvia l'esecuzione automatica della pipeline: inizio del *commit stage*
- Completato il *commit stage* con successo, gli *artifacts* vengono automaticamente storicizzati in un apposito repository
- Il server di *continuous integration* recupera gli *artifacts*, creati nello step precedente, dal repository e riproduce automaticamente un ambiente *production-like*
- Il server di *continuous integration* lancia automaticamente l'esecuzione degli *acceptance tests*, all'interno dell'ambiente di testing riprodotto
- Superando con successo i test automatici, l'applicazione viene marcata come idonea per la *release*
- La persona incaricata al testing manuale, può riprodurre manualmente l'ambiente di testing recuperando gli *artifacts* creati in seguito al *commit stage*. Eseguito il *manual testing*, l'applicazione prosegue il suo percorso di validazione

- Il server di *continuous integration* può, a questo punto, eseguire il software direttamente nell'ambiente di produzione. Tuttavia, esso ancora non è stato marcato come rilasciato (*released*) ma viene semplicemente sottoposto a dei test capacitivi
- A questo punto, in assenza di problemi, la produzione è approvata dai business manager che la dichiarano ufficialmente *released*

2.6 Conclusione

Avendo analizzato la struttura di una *deployment pipeline* per la *continuous delivery* del software, e avendone discusso le varie fasi di esecuzione, si evincono i notevoli vantaggi di questa tipologia di processo. Una *release* del software, automatizzata nelle sue fasi più critiche, che si pone come obiettivo quello di consegnare software di maggiore qualità al cliente, diminuendo il rischio di ottenere risultati diversi da quelli richiesti.

Capitolo 3

Implementazione della pipeline

3.1 Introduzione

In questo capitolo, verrà descritta in dettaglio l'implementazione pratica di una *Continuous Delivery* pipeline con l'obiettivo di validare e rilasciare in pochi minuti una semplice applicazione software. Il codice utilizzato, e discusso in seguito, è possibile sia consultarlo in Appendice sia scaricarlo dal *repository* GitHub cui si farà sempre riferimento[8].

3.2 Creazione e configurazione dell'ambiente

Per iniziare, abbiamo la necessità di configurare e avviare una *network* di macchine virtuali, al fine di simulare il comportamento reale delle *workstation* coinvolte nel flusso di esecuzione della pipeline. Il team dei *developers*, ad esempio, ha bisogno di un server Web su cui far girare l'applicazione da rilasciare, così come il team *operations* ha bisogno di un ambiente di testing dedicato all'esecuzione dei vari *test script*. Si necessita, dunque, di qualche strumento che simuli questo tipo di esigenze.

Si potrebbe pensare di utilizzare un software per la virtualizzazione quale Oracle VM VirtualBox[9]. Ciò richiederebbe di:

- Scaricare i file immagine dei sistemi operativi da virtualizzare
- Configurare **manualmente** ciascuna macchina virtuale, allocando risorse, gestendo periferiche o altre impostazioni
- Avviare, sospendere o arrestare **manualmente** una macchina per volta

[8]URL=https://github.com/marcopu/appTesi
[9]URL=https://www.virtualbox.org

L'errore di fondo, evidenziato dagli step precedenti, è il carattere manuale delle operazioni da eseguire. Quello che si desidera è poter riprodurre l'infrastruttura desiderata in pochi minuti, creando e arrestando l'ambiente in qualsiasi momento si voglia, evitando ogni tipo di configurazione manuale che possa risultare da *bottleneck* temporale nel flusso di esecuzione. Per questo motivo, anzichè utilizzare VirtualBox come software *stand-alone*[10], lo si utilizza come *provider* di un altro tool che consente di creare, configurare e gestire via codice una rete di macchine virtuali. Il tool in questione è Vagrant[11], uno degli elementi più importanti della *software-suite* targata HashiCorp.

Il tool Vagrant viene gestito interamente da linea di comando, con istruzioni che consentono di gestire il ciclo di vita delle macchine virtuali su principio dell'*Infrastructure as Code*[12]. Come requisito di funzionamento, Vagrant necessita di un *provider* di virtualizzazione compatibile, che sia VirtualBox oppure VMWare, su cui poggeranno le macchine virtuali che andremo a creare. Queste ultime vengono definite e configurate all'interno di uno script con sintassi Ruby, denominato Vagrantfile, generabile da *command-line* grazie al comando **`vagrant init`**. Un esempio di configurazione all'interno del Vagrantfile viene qui riportato:

```
config.vm.define "Esempio" do |esempio|
  esempio.vm.box = "BOX"
  esempio.vm.network "forwarded_port", guest: 80, host: 8080
  esempio.vm.network "private_network", ip: "ADDRESS"
  esempio.vm.provision "PROVISION"

  esempio.vm.provider "PROVIDER" do |vb|
      vb.name = "Esempio"
      vb.memory = 2048
      vb.cpus= 2
  end
end
```

Con questa sintassi si va a definire una macchina virtuale denominata **`Esempio`** - identificata dalla variabile **`esempio`** - cui assegniamo l'indirizzo IP **`ADDRESS`** all'interno di una rete privata; nulla vieta di definire una **`public_network`**, anzichè una

[10] «In informatica, l'espressione *stand-alone* indica che un oggetto o un software è capace di funzionare da solo o in maniera indipendente da altri oggetti o software, con cui potrebbe altrimenti interagire.» (Wikipedia)

[11] URL=https://www.vagrantup.com

[12] «*Infrastructure as Code* (IaC) è il processo di gestione e *provisioning* dei computer attraverso file di definizione leggibili dalla macchina.» (Wikipedia)

private_network come in questo caso. Oltre a definire un *port-mapping* tra la porta 8080 dell'host e la porta 80 della macchina guest, viene definito il **BOX** e il tipo **PROVISION** di *provisioning* utilizzato. I *Vagrant boxes* sono dei *packages* contenenti l'immagine della macchina virtuale da voler riprodurre. Per quanto riguarda il *provisioning*[13] è possibile specificarne la modalità: sia tramite uno *shell-script* sia tramite stumenti di *orchestration*[14] quali Puppet, Chef o Ansible, è possibile eseguire comandi di installazione e configurazione sulla macchina virtuale creata. Digitando da *command-line* il comando **vagrant up**, il *box* scelto viene scaricato dal *Vagrant cloud*[15] e ne viene eseguito il *provisioning* con la modalità definita. La macchina virtuale sarà, a questo punto, in esecuzione all'interno del **PROVIDER** definito nel Vagrantfile, disponendo di due CPU e di una memoria RAM pari a 2048 MB (nell'esempio riportato). Con i comandi **vagrant suspend** e **vagrant halt** è possibile, rispettivamente, sospendere e arrestare le macchine, mentre con un **vagrant destroy** viene eliminato il Vagrantfile e le configurazioni ad esso associate. Con l'esecuzione di semplici comandi, e l'utilizzo di uno script, è possibile dunque creare, gestire e distruggere risorse virtuali, senza la necessità di installare o configurare manualmente ogni loro componente.

Nel nostro caso, disponiamo di un sistema host Windows, sul quale viene generato inizialmente un Vagrantfile da *console* (vedi Appendice A.1). Al suo interno vengono definite e configurate, con codice simile a quello analizzato in precedenza, delle macchine virtuali, definendo per ognuna un certo *provisioning*. In particolare è stata definita una macchina virtuale per simulare ciascuno dei seguenti ruoli:

- Un server di *Continuous Integration* che si occupa di orchestrare i diversi *tasks* all'interno dell'infrastruttura creata
- Un server *LAMP*[16] su cui l'applicazione è in esecuzione
- Un server dedicato al testing, riproducendo un ambiente *production-like*

Nel corso della discussione, aggiungeremo ulteriori funzionalità con l'obiettivo di incrementare l'efficienza della pipeline.

[13] URL=https://www.vagrantup.com/docs/provisioning
[14] Coordinamento e gestione di architetture e/o servizi
[15] URL=https://app.vagrantup.com/boxes/search
[16] Acronimo che indica una piattaforma software per lo sviluppo di applicazioni Web che prende il nome dalle iniziali dei componenti software con cui è realizzata: **L**inux **A**pache **M**ySql **P**hp

3.3 Implementazione della pipeline in Jenkins: *upstream* e *downstream jobs*

Per quanto riguarda la macchina che ospita il server di *Continuous Integration*, utilizzeremo lo pseudonimo **`JenkinsVM`**. Per la sua installazione è stato utilizzato il *box* **`ubuntu/xenial64`** da *Vagrant Cloud*, il quale prevede l'installazione del sistema operativo Ubuntu 16.04 LTS (Xenial). Per quanto riguarda il *provisioning*, è stato utilizzato lo script **`jenkins.sh`** (vedi Appendice A.1.1), tramite cui:

- Viene installato Oracle Java Version 8, sia JRE che JDK
- Viene installato Jenkins, impostando il server di *Continuous Integration* in ascolto sulla porta http 8282
- Viene installato Maven per le dipendenze

Con l'esecuzione da *console* del comando **`vagrant up`**, viene completato il *provisioning* della JenkinsVM. Collegandosi al server sulla porta che è stata definita[17], siamo in grado di accedere alla *Jenkins User Interface* e personalizzare il nostro *automation server* per *build*, *test* e *deploy* dell'applicazione. L'applicazione di cui parliamo è stata sviluppata in JSP[18], adottando un approccio TDD (*test driven development*), con delle classi di test definite prima della scrittura del codice da testare. Dotata di interfaccia grafica pressochè inesistente, essa consente ai dipendenti dell'azienda la creazione di ticket consultabili da tutte le risorse interne al dominio. Il focus del lavoro non è stato incentrato sullo sviluppo dell'applicazione, bensì sul validare e rilasciare l'applicazione stessa in maniera efficiente.

A questo punto, cominciamo a capire quali step compiere per costruire e configurare la nostra pipeline. In questa prima sezione si procederà con la creazione - tramite la *Jenkins UI* - di vari job, cui verranno delegate le funzionalità previste da ciascuno stage analizzato nel capitolo precedente. Il flow di esecuzione che andremo a strutturare è quello riportato nella Figura 3.1. Con la configurazione dei job che discuteremo e con la vasta gamma di *plugin* che Jenkins mette a disposizione, andremo ad automatizzare il processo di *deployment* dell'applicazione.

[17] La porta è modificabile in **`/etc/default/jenkins`**

[18] **J**ava **S**erver **P**ages - Tecnologia che consente la creazione di applicazioni di facile sviluppo, accesso e impiego per il Web.

Figura 3.1. Schema di *Continuous Deployment*

3.3.1 *Build* e *Packaging*

Il primo step da compiere è quello di creare un job per la compilazione e il *packaging* dell'applicazione. In Jenkins è possibile creare un nuovo item o progetto direttamente da interfaccia grafica (Figura 3.2); una volta che il job risulterà nella *dashboard* della pagina iniziale, sarà possibile configurarne i vari parametri.

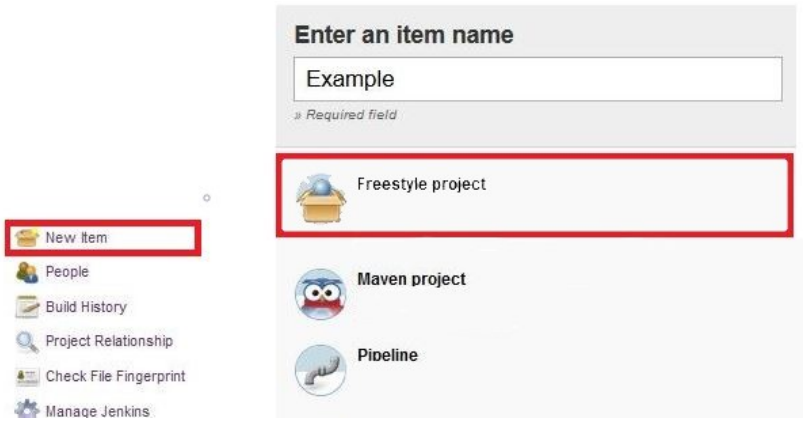

Figura 3.2. Creazione di un nuovo job in *Jenkins*

Il job in analisi è quello che è stato denominato **`Packaging`** e prevede i seguenti step:

- Innanzitutto, essendo questo job l'*entry-point* della nostra pipeline, specifichiamo nella sezione *Source Code Management* - tra le impostazioni di configurazione - il tipo di *version control system* che si utilizza, ossia Git - e l'URL del *repository* che traccia l'intera evoluzione del progetto.

- Nella sezione *Build* invece, definiamo la lista di comandi da eseguire. In seguito a qualunque tipo di *commit* sul sistema di *version control*, il job Packaging comincerà la sua esecuzione ed eseguirà una serie di comandi *shell* specificati nella relativa sezione. Con i comandi

```
mvn clean -DskipTests
mvn compile -DskipTests
mvn package -DskipTests
```

  l'applicazione viene compilata e archiviata in un archivio WAR[19]. Con l'opzione **`DskipTests`** si omette, in questa fase, l'esecuzione dei test unitari definti nelle classi di test del progetto.

- Infine, nella sezione *Post-buil actions*, si procede con l'archiviazione di tutti gli *artifacts* relativi al progetto, i quali verranno poi utilizzati negli stage successivi della pipeline. Viene specificato, inoltre, il *downstream* job da eseguire a valle di quello corrente.

3.3.2 *Orchestrating*

Il primo *downstream* job che viene definito è il job **`Orchestrating`**. Esso ha il compito di orchestrare i vari *tasks* all'interno della pipeline, facilitando il *deployment* dell'applicazione e la *configuration management*. Ci sono diversi tool *open source* in grado di ricoprire questo ruolo, tra cui Puppet, Chef e Ansible. Alla base del loro funzionamento vi sono dei file, denominati *cookbooks* o *playbooks* a seconda del tool scelto, in cui vengono definite le operazioni e i comandi da eseguire. In questo laboratorio, è stato scelto Ansible per la sua struttura *agentless*[20] - a differenza degli altri due - che consente di snellire l'infrastruttura e di ridurre possibili *overhead* a livello network. E' stato deciso, quindi, di creare e configurare con Vagrant una nuova macchina virtuale dedicata all'installazione di Ansible. La macchina, denominata **`AnsibleVM`**, presenta le stesse caratteristiche hardware e software della JenkinsVM, ovviamente con configurazioni di rete e *provisioning* - tramite lo *shell-script* **`ansible.sh`** (vedi Appendice A.1.1) - differenti. Entrando nei dettagli del job

[19] « ***W***eb *application* ***AR***chive - E' un archivio usato in Java per raggruppare diversi tipi di file che danno vita ad un'applicazione Web. Viene usato dai programmatori Java proprio per distribuire tutto l'applicativo software sviluppato.» (Wikipedia)

[20] Tipo di architettura in cui sui vari nodi non viene richiesta l'installazione di software specifico per la comunicazione con il proprio controllore. L'utilizzo di un server SSH, in questo caso, è più che sufficiente.

Orchestrating, nella rispettiva sezione *Build* non ci limitiamo a eseguire dei comandi *shell* ma utilizziamo i seguenti *plugin*[21]:

- *Copy Artifact Plugin* - Grazie al suo utilizzo, il job Orchestrating è in grado di copiare gli *artifacts* dal progetto Packaging in *upstream* (Figura 3.3). Gli *artifacts*, all'atto del salvataggio, vengono storicizzati da Jenkins nel proprio *workspace*, in cui viene tracciata ogni *build* di ogni progetto.

Figura 3.3. Utilizzo del plugin *Copy Artifact*

- *Publish Over SSH Plugin* - Esso ci consente di trasferire un set di file o eseguire comandi tramite il protocollo SSH (Figura 3.4). Il server di destinazione deve essere registrato nelle impostazioni generali di Jenkins, il quale stabilirà la connessione SSH all'atto dell'utilizzo del *plugin*.

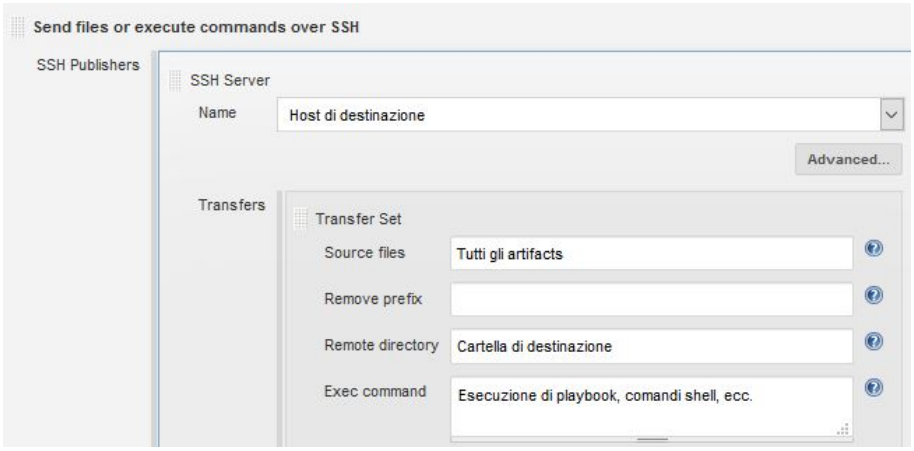

Figura 3.4. Utilizzo del plugin *Publish Over SSH*

Il job Orchestrating trasferisce innanzitutto gli *artifacts* - copiati dal progetto Packaging in *upstream* - sulla AnsibleVM tramite il secondo *plugin*, il quale opera via

[21]In *Jenkins*, l'installazione di un *plugin* è un'operazione immediata. Accedendo al pannello generale delle impostazioni, è necessario cliccare sulla voce relativa alla gestione dei *plugin* per cercare e installare l'estensione desiderata

SSH. Dopodichè Jenkins, nuovamente tramite SSH, esegue sulla AnsibleVM il *playbook* specificato. In generale, la sintassi del comando di esecuzione di un *playbook* è la seguente:

```
ansible-playbook PLAYBOOK
```

In cui a **PLAYBOOK** bisogna sostituire il file desiderato. I *playbooks* di Ansible richiedono una sintassi ben precisa, di tipo YAML[22]: ciascuno di essi contiene una serie di *plays* - ossia *tasks* da eseguire sull'host specificato - che richiamano moduli Python. Per una approfondita documentazione circa la vasta gamma di moduli che compongono un file YAML, si rimanda alla *Ansible Documentation*[23]. I moduli più utilizzati in questo lavoro, sono stati il modulo **hosts** e il modulo **tasks**: nel primo, si specificano gli host su cui verranno eseguite le operazioni definite nel secondo. Sulla macchina AnsibleVM, è possibile editare il file **/etc/ansible/hosts** aggiungendo nuovi host e relativi indirizzi (vedi Appendice A.1.2). Un esempio generico di *playbook* è il seguente:

```
- hosts: nodo1, nodo2, ..., nodoN
  tasks:
    - name: Task #1
      command: do something

    - name: Task #2
       shell: do something more by shell
```

Con il quale, intuitivamente, i **tasks** - e relativi comandi - verranno eseguiti sui nodi specificati nel modulo **hosts**.

L'utilizzo dei *playbooks* di Ansible può essere visto come una sorta di *provisioning on demand* dell'intera infrastruttura. Ambienti *ad hoc* di sviluppo, testing e *deployment* vengono configurati in tempo reale nel corso di esecuzione della pipeline, evitando installazioni e configurazioni massive nel *provisioning* effettuato all'avvio di ciascuna macchina. Nel nostro caso, ad esempio, abbiamo bisogno di Apache Tomcat[24] per l'esecuzione della nostra applicazione in fase di sviluppo. Esso deve essere installato sulla **DevelopmentVM**, ossia il server *LAMP* la cui configurazione e il *provisioning by shell* sono definiti nel Vagrantfile (vedi **lamp.sh** in Appendice A.1.1). Si può scegliere, dunque, di includere l'installazione di Apache Tomcat:

[22]Documentazione ufficiale *YAML Syntax*

[23]URL=http://docs.ansible.com/ansible/latest/list_of_all_modules.html

[24]Piattaforma software *open source* che implementa le specifiche JSP e *servlet*, consentendo l'esecuzione di applicazioni Web sviluppate in linguaggio Java.

- Nel *provisioning* alla creazione della macchina DevelopmentVM
- In un *playbook* che esegue - quando richiesto da Jenkins - le operazioni di installazione e configurazione del *servlet container*

Supponiamo di essere nel primo caso e che, per qualche ragione, venga richiesto che l'applicazione debba essere eseguita in un *servlet container* diverso da Apache Tomcat, che sia GlassFish oppure JBoss. L'unica possibilità, per soddisfare i nuovi requisiti, è quella di dovere riconfigurare completamente la macchina. Nel secondo caso, la soluzione sarebbe ben più immediata e flessibile - dovendo semplicemente modificare un *task* - evitando di impattare in maniera significativa sull'esecuzione della pipeline. Pertanto, il job Orchestrating si occupa di installare (qualora non fosse presente) e avviare l'esecuzione di Apache Tomcat sulla DevelopmentVM, specificando tale macchina nel modulo **`hosts`** del *playbook* **`starttomcat.yml`** (vedi Appendice A.2).

3.3.3 *Developing*

Al termine del job Orchestrating, avremo Apache Tomcat in stato *running* sulla porta 8080 http della nostra DevelopmentVM; lo step successivo sarà quello di eseguire la nostra applicazione *app* all'interno del *servlet container*. A tal scopo, viene creato in Jenkins un nuovo job - in *downstream* al job Orchestrating - denominato **`Development`**. Esso dovrà:

- Copiare gli *artifacts*, tramite il plugin *Copy Artifact* descritto in precedenza
- Trasferire, tramite il plugin *Publish Over SSH* descritto in precedenza, l'archivio **`app.war`** dalla *JenkinsVM* alla sottocartella **`/opt/tomcat/webapps`** della DevelopmentVM, sulla quale è installato e avviato Apache Tomcat.

Al termine del job Development, collegandoci all'indirizzo IP della DevelopmentVM sulla porta 8080 - specificando nella URL il *context* **`/app`** - troveremo in esecuzione la nostra applicazione.

3.3.4 *Unit Testing*

Proseguendo con la costruzione della pipeline in versione *job-chain*, lo step successivo prevede l'esecuzione degli *unit tests* definiti in fase di sviluppo (vedi Appendice A.4.1). Viene creato, a tal fine, un nuovo job che denominiamo **`UnitTesting`**. Utilizzando nuovamente la logica dei *playbooks*, Jenkins comanderà - tramite SSH - l'esecuzione di un nuovo *playbook* sulla AnsibleVM. Il *playbook* **`unit.yml`** (vedi

Appendice A.2) esegue fondamentalmente il comando **`mvn test`**, il quale utilizzerà il *framework* JUnit di Java per l'effettiva esecuzione dei test unitari, definiti nelle classi di test - **`AttachmentTest.java`** e **`TicketTest.java`** (vedi Appendice A) - del progetto. Da sottolineare che il *framework* JUnit viene richiamato da Maven, in seguito all'esecuzione del comando **`mvn test`**, grazie alla *dependency*

```
<dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>4.5</version>
     <scope>test</scope>
</dependency>
```

definita nel file di configurazione **`pom.xml`** (vedi Appendice A.3).

3.3.5 *Code Analysis*

Oltre all'esecuzione degli *unit tests*, per concludere il *commit stage* della nostra pipeline, andiamo ad effettuare un'analisi di qualità del codice creando - in *downstream* al job UnitTesting - un nuovo job che denominiamo **`CodeAnalysis`**. Per la cosiddetta *Continuous Code Quality*, è stato utilizzato SonarQube[25] - tool *open source* sviluppato in Java - che consente di analizzare la qualità del codice sorgente, rilevandone *bugs*, vulnerabilità e *code smell*[26]. L'utilizzo di SonarQube prevede l'installazione e la configurazione di un SonarQube Server (vedi Appendice A.3.1). Pertanto, è stato deciso di utilizzare una quarta macchina - che denominiamo **`TestingVM`**[27] e con *provisioning by shell* in **`sonar.sh`** (vedi Appendice A.1.1) - su cui SonarQube è in esecuzione (porta http 9000). Infine, aggiungendo nel file **`pom.xml`**:

- Il *plugin* di SonarQube per Maven

```
<plugin>
     <groupId>org.sonarsource.scanner.maven</groupId>
     <artifactId>sonar-maven-plugin</artifactId>
     <version>3.3.0.603</version>
</plugin>
```

[25] URL=https://www.sonarqube.org

[26] Letteralmente "puzza del codice", indica debolezze di progettazione (codice duplicato, metodi troppo lunghi, ecc.) che riducono la qualità del codice e incrementano il *technical debt*.

[27] Utilizziamo il *Vagrant Box* **`ubuntu/xenial64`**, così come per JenkinsVM, AnsibleVM e DevelopmentVM.

- Le proprietà dell'host su cui è in esecuzione SonarQube

```
<properties>
     ...
     <sonar.host.url>
        http://192.168.110.40:9000/sonar
     </sonar.host.url>
     ...
</properties>
```

è possibile utilizzare il comando **`mvn sonar:sonar`** per condurre l'analisi di qualità dell'applicazione. Nuovamente, questo comando verrà inserito in un *Ansible playbook* - denominato **`analyzer.yml`** (vedi Appendice A.2) - eseguito da Jenkins sulla AnsibleVM. Il risultato dell'analisi sarà consultabile nella *dashboard* di SonarQube alla URL specificata nelle propretà (Figura 3.5).

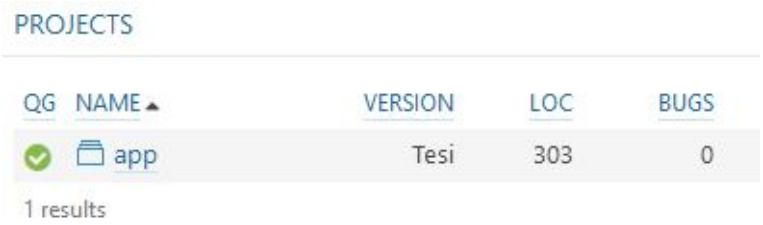

Figura 3.5. *Quality Gate* - QG - del progetto analizzato con *SonarQube*

3.3.6 *Automated Testing*

Arrivati a questo punto, l'applicazione *app* è in esecuzione sulla DevelopmentVM in Apache Tomcat, gli *unit tests* sono stati effettuati e l'analisi di qualità del codice è stata condotta. Con l'esecuzione dei *playbooks* di Ansible, orchestrati puntualmente da Jenkins, vengono eseguiti i vari *tasks* relativi a ciascuno stage della pipeline, dall'esecuzione alla validazione dell'applicazione. Quello che manca è uno degli step fondamentali, ossia l'esecuzione dei *functional acceptance tests*. Tra i diversi tool a disposizione è stato scelto Selenium[28], una delle migliori soluzioni per l'*automated testing*. Qualsiasi browser, tra quelli più diffusi, è in grado di supportarlo, prestandosi in maniera ottimale all'esecuzione di *automated tests* su una applicazione Web. L'utilizzo di Selenium riduce i margini di errore relativi all'esecuzione di una applicazione e ne aumenta la portabilità, dato che le *test suites* utilizzate consentono

[28]URL=http://www.seleniumhq.org/docs

di validare l'esecuzione dell'applicazione su più browser o framework, in un unico passaggio.

Ci sono diverse modalità in cui Selenium può essere utilizzato. Una prima alternativa può essere *Selenium RC*[29], il quale consiste nell'installazione di un *Selenium Server*. Esso è in grado di interagire con i diversi browser, inoltrando - in formato JavaScript - i comandi di testing al *Selenium Core* abilitato su di essi (Figura 3.6).

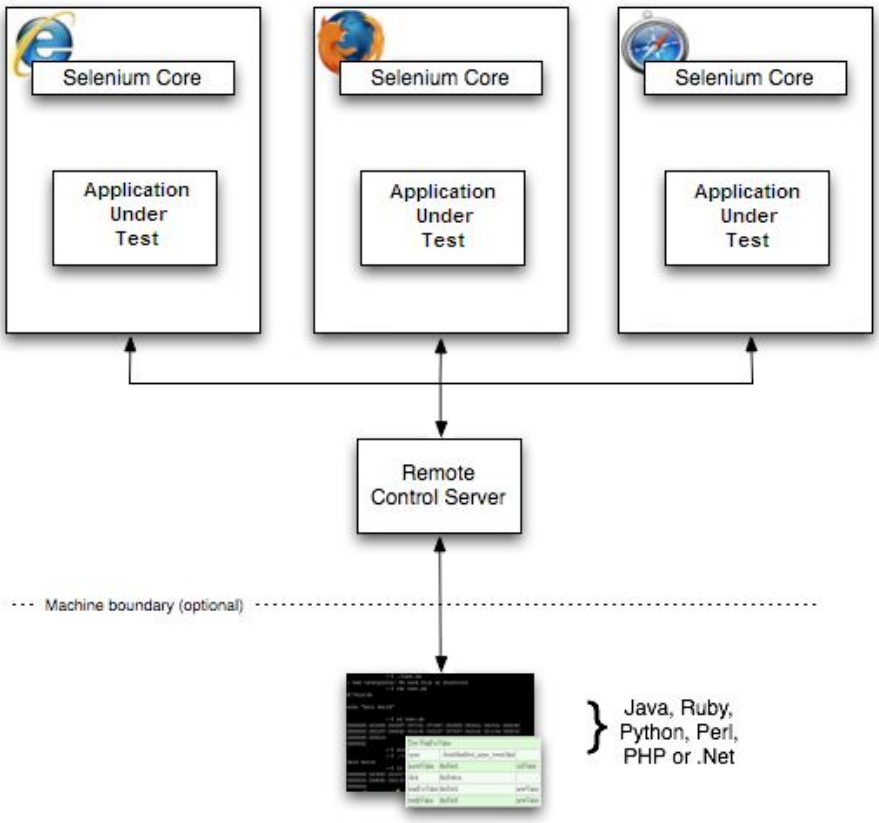

Figura 3.6. Funzionamento di *Selenium RC*

Tale soluzione però richiede un doppio passaggio attraverso il *Selenium Server* (che funge da ponte), limitando la velocità di esecuzione degli *automated tests* non estremamente veloci per natura. Per questo motivo, è stato evitato l'utilizzo di un ulteriore server per ospitare Selenium. La soluzione adottata è stata quella di utilizzare - anche per questo stage - la TestingVM configurata in precedenza, semplicemente installando delle Python API e alcuni *driver*, necessari a Selenium per interfacciarsi con i browser su cui l'applicazione viene testata. In particolare, da terminale:

[29]URL=http://www.seleniumhq.org/docs/05_selenium_rc.jsp

- Vengono installati i pacchetti relativi a Selenium con il comando **pip install selenium**
- Vengono installati i seguenti driver: **chromedriver** per interfacciarsi a Google Chrome, **geckodriver** per interfacciarsi a Mozilla Firefox (vedi Appendice A.4.2)

Completata la procedura di installazione e configurazione di Selenium, siamo pronti alla stesura dei test. La *test suite* utilizzata - **suite.py** (vedi Appendice A.4.2) - definisce alcuni casi di test, da eseguire sui browser Mozilla Firefox e Google Chrome, al fine di verificare il corretto *behaviour* dell'applicazione. Essi vanno a simulare alcune operazioni che l'utente esegue in fase di utilizzo dell'applicazione, sfruttando anche il layout strutturale delle pagine Web. Ad esempio, la seguente definizione

```
def test_openChrome(page):
        driver = page.driver
        driver.get("http://192.168.110.30:8080/app/tickets")
        page.assertIn("Consoft", driver.title)
```

simula le seguenti operazioni:

- Apertura di una pagina Web in Google Chrome
- Collegamento alla pagina **tickets** della nostra *app*, in esecuzione all'interno di Apache Tomcat sulla DevelopmentVM
- Verifica che il tag **<title>** di quella pagina corrisponda alla stringa "Consoft"

Senza l'utilizzo di interfacce grafiche, si va a riprodurre all'interno di un buffer virtuale - utilizzando la libreria Python **pvirtualdisplay** - una istanza del browser specificato, simulando varie operazioni che l'utente può eseguire interagendo con l'applicazione. Utilizzare frame virtuali *by code*, anzichè riprodurre interfacce grafiche, velocizza ulteriormente l'esecuzione di questa tipologia di test, incrementando performance ed efficacia. Anche in questo caso, l'esecuzione del *task* di testing viene gestito da un *Ansible playbook*, denominato **functional.yml** (vedi Appendice A.2). Il nuovo job **FunctionalTesting** - in *downstream* al job UnitTesting - viene dunque creato e configurato in Jenkins. Esso eseguirà - sulla TestingVM via SSH - lo script Python contenente i test, stampando[30] un report finale in formato HTML. Esso funge da feedback sulla corretta esecuzione dei test ed è consultabile nella directory **/reports** della TestingVM. Qualora tutti i test siano stati superati con successo, l'applicazione diviene un candidato ideale alla *release* finale.

[30]Per la stampa del report viene utilizzato lo script **HTMLTestRunner.py** (vedi Appendice A.4.2)

3.3.7 *Pipeline View* e considerazioni

Tutti i job creati e configurati finora sono visibili nella *Dashboard* di Jenkins (Figura 3.7).

Figura 3.7. La *Dashboard* di *Jenkins*

Concatenati tra loro, essi compongono la *deployment pipeline* descritta teoricamente nel capitolo precedente. Jenkins consente anche di creare una *view* completa della pipeline in esecuzione, grazie all'installazione del *Build Pipeline Plugin* oppure del *Delivery Pipeline Plugin* (Figura 3.8). Essi differiscono da un punto di vista grafico ma entrambi visualizzano il numero di *build*, i vari stage della pipeline, il loro tempo di esecuzione e il loro stato di successo o fallimento.

Concatenare *upstream* e *downstream jobs* ci consente, dunque, di raggiungere il nostro fine primario: la costruzione e l'esecuzione della pipeline. Tuttavia:

- La configurazione dei vari stage, tramite la *Jenkins UI*, non è depositabile all'interno di un *version control system*. Supponiamo che la JenkinsVM venga eliminata e che Jenkins debba essere installato su un nuovo server per la *Continuous Integration*; procedendo nel modo finora raccontato, bisognerebbe ripopolare la *Dashboard* di Jenkins configurando nuovamente ogni job della *chain*. Si tratterebbe di una azione manuale che - seppure semplice da eseguire tramite *User Interface* - un pò stonerebbe all'interno di una infrastruttura automatizzata e sincronizzata come quella descritta in precedenza. Come è possibile trasformare tutto ciò in una *Infrastructure as Code* a tutti gli effetti?

Figura 3.8. *Pipeline View* tramite il plugin *Delivery Pipeline*

- Jenkins richiede che l'intera infrastruttura e le varie macchine siano avviate, configurate e disponibili nel momento in cui opera su di esse. Perchè, allora, non consentirgli di avviare *on demand* le risorse di cui necessita e, soprattutto, nel momento in cui ne necessita? In altre parole: come possiamo trasferire a Jenkins il controllo di Vagrant, al fine che sia lui a decidere quando e quali macchine avviare in un momento specifico?
- Infine, abbiamo visto che un ambiente *production-like* viene riprodotto - dal team di sviluppo - sulla DevelopmentVM, per l'esecuzione dell'applicazione. Anche i *functional acceptance tests*, eseguiti dal job FunctionalTesting, sono stati eseguiti sull'applicazione in esecuzione - sulla DevelopmentVM - all'interno di Apache Tomcat. Perchè non riprodurre - anche in fase di testing - un ambiente *production-like* che non vada ad interferire con quello di sviluppo, parallelizzando *development* e *testing*?

Queste domande trovano risposta nell'ottimizzazione del processo di cui discuteremo nella prossima sezione.

3.4 Ottimizzazioni della pipeline

3.4.1 Docker scende in campo: *development* e *testing* in parallelo

L'esecuzione dei test funzionali all'interno dell'ambiente di *development*, non è sicuramente una buona prassi. Il processo di validazione deve essere condotto in un

ambiente indipendente da quello in cui gli sviluppatori eseguono l'applicazione dopo averla compilata, al fine di parallelizzare lo sviluppo e il testing dell'applicazione stessa. In nostro soccorso giunge uno dei tool più utilizzati nell'ambito della virtualizzazione leggera: Docker[31]. Il suo utilizzo ci consente di creare un *production-like environment* in cui eseguire i nostri test. Innanzitutto, all'interno del Vagrantfile di partenza, viene definita una nuova macchina - denominata **`DockerVM`** - utilizzando il *Vagrant box* **`kreator/trusty64-docker`**[32], con sistema operativo Ubuntu Trusty x64 e con Docker già configurato su di esso. Il compito di questa nuova macchina è stato quello di creare una *Docker image*[33] a partire dal **`Dockerfile`** definito (vedi Appendice B.3).

Tramite il comando

```
docker build PATH_DOCKERFILE
```

viene costruita l'immagine di Apache Tomcat[34] con al suo interno l'archivio **`app.war`** dell'applicazione in esecuzione. Successivamente alla sua creazione, l'immagine viene registrata sul **`docker registry`** locale della DockerVM, tramite il comando

```
docker push localhost:5000/app
```

A questo punto, dalla macchina TestingVM è possibile effettuare il **`pull`** dell'immagine ed eseguire il *container*[35] per il testing, tramite il comando **`docker run`**. L'ambiente di testing sarà, dunque, in esecuzione e su di esso potranno essere effettuati i test funzionali. La serie di *tasks* descritti finora, compongono un nuovo *Ansible playbook* - denominato **`dockerplay.yml`** (vedi Appendice B.3) - eseguito da Jenkins al fine di configurare l'ambiente in cui eseguire gli *automated tests*.

3.4.2 Nodi *master* e *slave* in Jenkins

Come abbiamo visto finora, la JenkinsVM ha avuto sempre un ruolo fondamentale, quello di coordinare i *tasks* da eseguire all'interno dell'infrastruttura virtuale. Tuttavia, la soluzione ottimale sarebbe quella in cui l'avvio dell'infrastruttura stessa sia sotto il controllo di Jenkins. Vorremmo, dunque, che Jenkins avesse il controllo su Vagrant, responsabile dell'avvio e della configurazione di AnsibleVM, DevelopmentVM, DockerVM e TestingVM. Così facendo, tali macchine saranno utilizzate

[31] URL=https://www.docker.com/what-docker

[32] Scaricabile dal *Vagrant Cloud* (URL=https://app.vagrantup.com/boxes/search)

[33] E' l'equivalente del box in Vagrant, con la differenza che la virtualizzazione avviene a livello *kernel* del sistema operativo. Pertanto si parla, nel caso di Docker, di virtualizzazione leggera.

[34] URL=https://hub.docker.com/_/tomcat/

[35] La singola istanza di una *Docker image* prende il nome di *container*.

esclusivamente nel momento in cui la pipeline in esecuzione richiede le loro risorse.

Pertanto, il Vagrantfile iniziale, che conteneva la definizione di tutte le macchine virtuali, è stato diviso in due parti:

- Vagrantfile *master* - In cui viene definita la macchina ospitante il nodo master di Jenkins (vedi Appendice B.1)
- Vagrantfile *slave* - In cui vengono definite tutte le altre macchine virtuali appartenenti all'infrastruttura (vedi Appendice B.2)

Entrambi i file, nel nostro caso, risiedono sulla macchina host Windows. Come prima cosa, tramite il comando **`vagrant up`**, viene avviato il server di *Continuous Integration* - ossia la JenkinsVM - dal Vagrantfile *master*. A questo punto, bisogna consentire a Jenkins - in esecuzione su un server virtuale di tipo Ubuntu - di avviare le altre macchine dal Vagrantfile *slave* presente sull'host Windows. Ecco dunque che, accedendo alla pagina iniziale del *Jenkins master*, è stato creato e configurato un nuovo nodo (Figura 3.9) - denominato **`winslave`** - il quale non è altro che uno *slave-agent* cui il *master* delega l'esecuzione di alcuni comandi. Sulla macchina host Windows è stato scaricato il file JAR **`agent.jar`** ed eseguito un comando *bash* del tipo

```
start java -jar agent.jar -jnlpUrl http://master_address
```

grazie al quale viene instaurata la comunicazione *master-slave*. Questo comando è stato inserito in uno script - **`myscript.bat`** (vedi Appendice B.1) - che viene eseguito successivamente al **`vagrant up`** del nodo *master*.

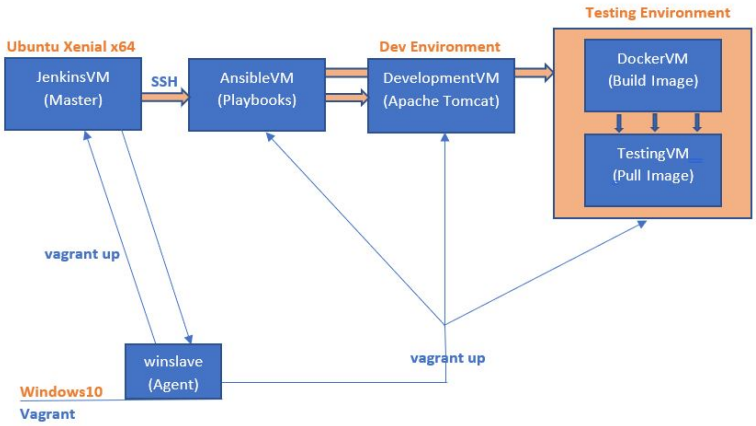

Figura 3.9. Creazione del nodo *winslave* sull'host Windows

3.4.3 La *Declarative Pipeline* di Jenkins

Il modo ulteriore per definire una pipeline in Jenkins, anzichè creare una serie di job concatenati tra loro, è quello di creare uno script - con sintassi Groovy - gestibile in maniere differenti:

- Viene creato uno **`Jenkinsfile`** (vedi Appendice B.4) contenente lo script per la creazione della pipeline. Questo file viene caricato nel nostro sistema di *version control* e utilizzato non appena l'esecuzione della pipeline viene innescata. In Jenkins, stavolta, viene configurato un nuovo e singolo *item* di tipo *Pipeline* (Figura 3.10), specificando semplicemente il *repository* GitHub in cui è memorizzato lo Jenkinsfile. Automaticamente, i vari stage definiti al suo interno verranno eseguiti.

Figura 3.10. Creazione di un nuovo *item* di tipo *Pipeline*

- Anzichè memorizzare lo script Groovy sul *repository*, esso viene inserito direttamente nell'apposito editor della *Jenkins User Interface*, tra le impostazioni di configurazione del progetto di tipo *Pipeline* (Figura 3.11).

Utilizzare uno Jenkinsfile, condiviso da tutti tramite *repository*, è sicuramente il valore aggiunto che cercavamo. Gli stessi comandi utilizzati precedentemente per la configurazione dei vari job, vengono ora inseriti all'interno dello script, sortendo il medesimo effetto.

Figura 3.11. Script Groovy inserito all'interno del job *Pipeline*

Lo script Groovy utilizzato, evidenzia gli stage principali di una *Continuous Delivery pipeline*, nel seguente modo:

```
pipeline{
  agent any

  stages{
    stage('Stage'){
      steps{
         echo 'Esecuzione step1...'
         echo 'Esecuzione step2...'
                ...
         sh 'echo 'Esecuzione stepN...''
      }
    }
  }
}
```

- **`agent`** indica quale nodo di rete è coinvolto nell'esecuzione della pipeline
- **`stage`** definisce semplicemente lo stage della pipeline in cui devono essere eseguiti i comandi contenuti nel blocco **`steps`**
- **`sh`** indica l'esecuzione di comandi *shell*

Nel nostro caso, il primo stage della *Declarative Pipeline* si occupa di avviare lo *slave* Jenkins, configurato come descritto nella sezione precedente. Una volta che l'infrastruttura è in funzione e tutte le macchine configurate, si procede con i seguenti stage:

- Il *Packaging* dell'applicazione
- L'esecuzione degli *Unit Tests*

- L'analisi di qualità del codice (*Code Analysis*)
- Il *Development* dell'applicazione sulla DevelopmentVM
- L'esecuzione degli *acceptance tests* sulla TestingVM

In ognuno di questi stage, un ruolo fondamentale è ricoperto dagli *Ansible playbooks*, orchestrati via SSH[36] dal Jenkins *master*. Inoltre, avendo instaurato l'architettura *master-slave* nella modalità descritta in precedenza, nello script Groovy della nostra *Declarative Pipeline* è possibile esplicitare il **`node`** su cui ogni stage deve essere eseguito. Al termine dell'esecuzione della pipeline, il nodo *master* delegherà allo *slave* il compito di sospendere tutte le macchine definite nel Vagrantfile *slave*, in attesa di un nuovo *trigger* che inneschi una nuova esecuzione della pipeline. Si è preferito sospendere le macchine - **`vagrant suspend`** - anzichè arrestarle del tutto, in quanto l'arresto al termine di ogni esecuzione avrebbe determinato un *overhead* - temporale e computazionale - troppo elevato all'avvio della pipeline.

Abbiamo analizzato, dunque, i vantaggi di una *pipeline as code* di questo genere (Figura 3.12). Essa è implementabile in maniera istantanea, eseguibile al volo - e quando si necessita - indipendentemente dal nodo su cui il Jenkins *master* è in esecuzione. Presenta, inoltre, una elevata flessibilità, consentendo di:

- Parallelizzare i vari stage
- Formare *fork* o *join* a piacimento
- Utilizzare via codice la vasta gamma di *plugin* che venivano installati gestendo la pipeline tramite interfaccia utente.

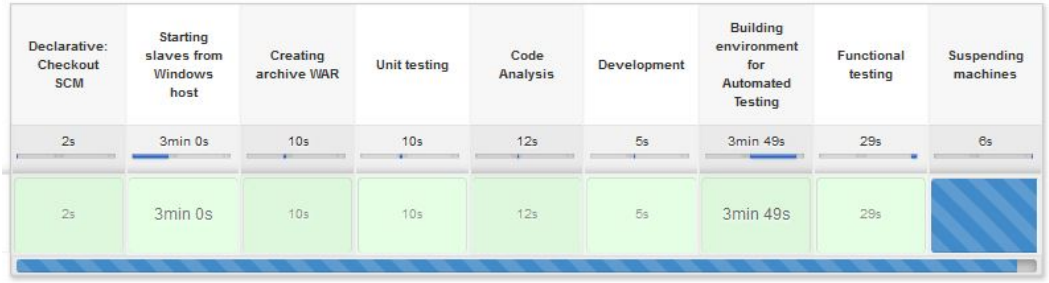

Figura 3.12. La *Declarative Pipeline*

[36]La comunicazione via SSH è basata sull'**`SSH Host Key Verification`**, ossia l'utilizzo di chiavi pubbliche e private per l'autenticazione della comunicazione *master-slaves*.

3.5 Conclusione

Siamo giunti alla conclusione di questo laboratorio, articolato in due fasi principali. In una prima fase, dopo aver spiegato le configurazioni dell'infrastruttura su cui andare a lavorare, indicando le varie macchine e i vari job definiti, si è lavorato molto sulla *User Interface* del tool Jenkins, estremamente funzionale ed estremamente completo nel fornire tutti gli strumenti necessari alla corretta implementazione di una pipeline per la *Continuous Delivery*. Al termine di questa prima fase, c'è stata una *revision* delle fragilità ancora presenti. Tali fragilità sono state completamente annullate con quanto svolto nella seconda fase del lavoro: la pipeline precedente, costituita interamente da job e *plugin*, è stata completamente trasformata in codice all'interno di uno script che ne ottimizza l'esecuzione e che ne consente una elevata flessibilità di configurazione. Parallelizzare, inoltre, la fase di testing a quella di sviluppo con l'utilizzo di *Docker containers*, consente di migliorare ulteriormente la qualità del software da rilasciare. Infine, il controllo quasi-totale da parte di Jenkins degli attori in gioco (*version control system*, *playbooks*, ecc.), è stato reso totale facendogli assumere il pieno controllo - in avvio e sospensione - dell'infrastruttura virtuale definita con Vagrant.

Nel prossimo capitolo, verrà confrontato il modello appena descritto con gli altri in uso - vecchi e nuovi - stimando, per ognuno di essi, tempistiche e qualità nel rilascio della stessa applicazione.

Capitolo 4

Analisi e confronti

4.1 Introduzione

In questa ultima parte, si procederà ad analizzare le prestazioni della soluzione di *Continuous Delivery* implementata nelle sezioni precedenti.

In particolare, si assegnerà un *maturity level* alla soluzione adottata, confrontando i vantaggi prestazionali a ciascun livello e discutendo i possibili margini di miglioramento che consentono di raggiungere il grado massimo possibile di ottimizzazione. Le stime che verranno riportate rispecchiano in maniera fedele l'aumento delle prestazioni, registrato grazie al passaggio da un livello all'altro. Sono molte, attualmente, le aziende che adottano soluzioni di questo genere, ognuna con logiche e tool più adatti alla propria tipologia di business. Ognuna, però, è in grado di raggiungere un proprio livello di maturità e consapevolezza, cercando di avere più o meno successo nella definizione di un processo di *release-automation*, volto a garantire la massima soddisfazione sia all'interno dei team sia del cliente finale.

4.2 Maturità della *Continuous Delivery*: dove siamo?

Così come accade per la realizzazione di ogni nuova tecnologia, diverse organizzazioni reagiscono in maniera differente all'adozione di una nuova soluzione per la *software-release* (*Continuous Delivery*) o di una nuova cultura organizzativa (*DevOps*). Di conseguenza, vi sono diversi livelli di avanzamento o *maturity levels* (Figura 4.1) che caratterizzano una evoluzione tutt'altro che immediata. [9]

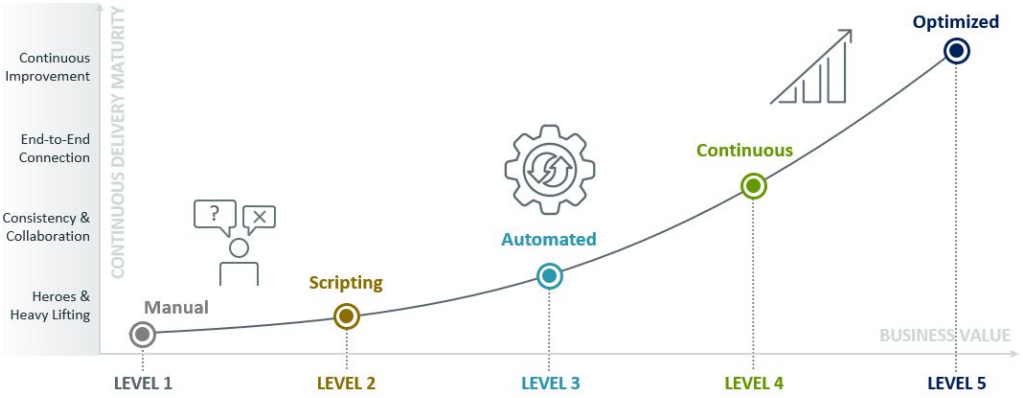

Figura 4.1. La *Continuous Delivery maturity* (Ravichandran et al., 2016)

- Livello 1 (*Manual*) - A questo livello si parla di silos organizzativi, ossia aree verticali distinte per competenza. Essi sono tali da ragionare in un'ottica di separazione, prediligendo l'ottimizzazione locale a scapito di quella globale, e richiedono che l'intervento manuale sia sempre necessario in ciascuna fase del processo di *delivery*. Ne derivano *releases* sempre meno frequenti e inclini ad errori in produzione, con conseguenze inevitabili sul business globale.
- Livello 2 (*Scripting*) - Ad ogni *release* viene ripianificato un nuovo processo, il quale viene tracciato in ogni sua fase. I vari team cominciano ad utilizzare un *version control system* per la condivisione del codice sorgente, automatizzando la fase di *build* dell'applicazione con l'utilizzo di script, i quali costituiscono l'unica componente *automated* all'interno del processo di *deployment*.
- Livello 3 (*Automated*) - L'intero processo di *release* viene monitorato e automatizzato, permettendo di riutilizzare lo stesso processo per rilasci continui e successivi dello stesso software. Tuttavia, viene utilizzato un *provisioning* già esistente, il quale riduce la scalabilità del *deployment*.

- Livello 4 (*Continuous*) - La *Continuous Delivery* del software promuove, all'interno della pipeline di *release-automation*, il rilascio incrementale del software, dallo sviluppo fino alla effettiva messa in produzione. L'*orchestration* del *provisioning-on-demand* e dei vari task, consente il *deployment* dell'applicazione in ambienti e su piattaforme differenti, incrementandone la qualità con rischio minore.
- Livello 5 (*Optimized*) - La pipeline diviene a tutti gli effetti un singolo punto di controllo: essa è in grado di gestire non solo la *release-automation* del software ma anche di distribuire, in maniera automatizzata e in base al carico di lavoro, la gestione dell'applicazione su più nodi o micro-servizi.

Non risulta difficile, dunque, collocare in corrispondenza del livello *Continuous* la soluzione implementata in questo elaborato. La *Continuous Delivery* pipeline, implementata e discussa nel capitolo precedente, non è dotata delle caratteristiche necessarie per classificarla totalmente come *Optimized*. Nel nostro caso, infatti, è stato aggiunto un grado di distribuzione minimo e pianificato, rappresentato dal nodo *slave* cui il *master* delega alcuni task. A livello *Optimized*, invece, la pipeline è in grado di sfruttare algoritmi complessi (es. *machine learning*) che le consentono di distribuire il processo di *release* su nuovi nodi o su quelli più adatti, massimizzando l'efficienza di ciascun team dell'organizzazione.

4.3 Le *Three Ways* del processo di maturità

Per fornire una misura ragionevolmente approssimata delle performance, è opportuno considerare ogni stage del processo di *release*, il quale ha un proprio peso specifico sulla misura finale. In questa sezione verranno fornite delle stime ad ogni livello di maturità della *Continuous Delivery* discussi in precedenza, considerando il rilascio di una applicazione di media complessità. Tuttavia, la notevolezza delle performance di una *release-automation* pipeline è di gran lunga più evidente all'aumentare della complessità del software da distribuire; in tal caso, avere rilasci di maggiore qualità in tempi minimi assume un valore esponenziale, dato che si ha a che fare con ambienti di sviluppo più articolati, *acceptance criteria* più specifici e ambienti *production-like* maggiormente ottimizzati.

Al tempo zero - Livello *Manual* - gli step manuali conferiscono al processo un carattere *waterfall*, con colli di bottiglia temporali in ciascuna delle sue fasi (Figura 4.2). In dettaglio, sono stati stimati mediamente:

- 45 giorni comprendenti analisi dei requisiti, pianificazione e rimodellizzazione dovuta ad errori o variazione dei requisit iniziali

- 55 giorni comprendenti lo sviluppo dell'applicazione e le modifiche successive al fallimento dei primi test, successivamente alla fase di *build*
- 30 giorni comprendenti l'esecuzione di test manuali, con continuo rimbalzo tra tester e sviluppatori
- 25 giorni per il rilascio in produzione, comprendenti la risoluzione di errori o configurazioni errate dell'ambiente

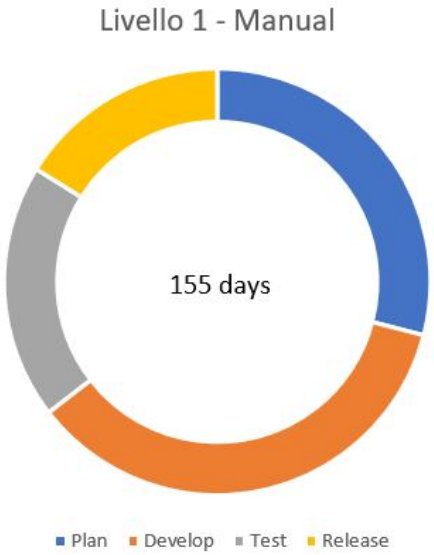

Figura 4.2. Livello *Manual*: le performance

A livello *Scripting* si comincia a delineare un primo miglioramento, rappresentato dall'utilizzo di script che impattano principalmente sulla fase di *build* e *test* dell'applicazione. L'assenza di una gestione automatizzata dei vari task e dei relativi feedback, però, non risolve i *bottleneck* in fase di progettazione iniziale e produzione finale. In dettaglio, vengono stimati:

- 43 giorni per la gestione dei requisiti e per la pianificazione del processo. Essa risente ancora della mancanza di un sistema di feedback continuo che consenta allo sviluppo di adattarsi alle nuove esigenze in tempo quasi-reale, senza dover intervenire continuamente sulla fase di design
- 37 giorni comprendenti lo sviluppo dell'applicazione e le modifiche necessarie a soddisfare nuovi requisiti e il superamento dei test

- 23 giorni per il testing dell'applicazione; quello che migliora è la velocita di esecuzione dei test grazie all'utilizzo di script che limitano la presenza umana in questa fase. Tuttavia, si continua ad avere uno stallo significativo dovuto all'attesa di un corretto completamento dello sviluppo
- 22 giorni per rilasciare il software in produzione, intervenendo manualmente su errori e configurazioni incompatibili

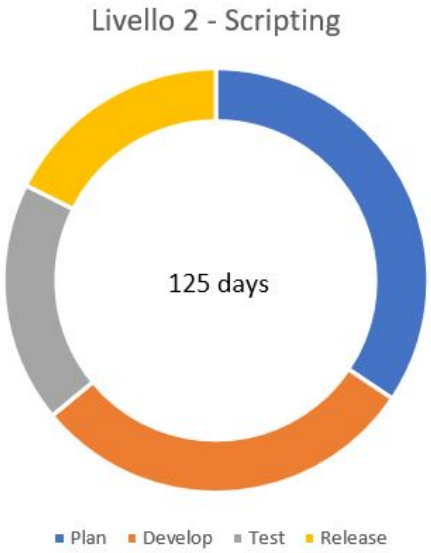

Figura 4.3. Livello *Scripting*: le performance

4.3.1 Step 1: da *Scripting* ad *Automated*

Il primo step - *First Way* - per una *Continuous Delivery* di successo, consiste nell'automatizzare il processo di *release* dallo sviluppo sino alla messa in produzione, migliorandone continuamente qualità ed efficienza. Adottare una strategia trasparente e collaborativa quale il *DevOps*, inoltre, conferisce al processo di *release automation* il controllo totale su ogni attività, garantendo l'interoperabilità delle varie fasi.

Precedentemente, in assenza di un processo automatizzato, si arrivava ad un paio di *deployment* per settimana dell'applicazione, per poi accorgersi, solamente in fase di rilascio in produzione, del funzionamento non corrispondente a quello desiderato. Ora, con un processo di *release-automation*, il *deployment* dell'applicazione viene effettuato almeno 50 volte per settimana, riducendo notevolmente il rischio di un

fallimento inaspettato in produzione. L'ingresso in campo della *Continuous Integration* alimenta, dunque, il *deployment* continuo dell'applicazione prima del rilascio definitivo, facendo registrare un netovole incremento delle performance.

In particolare, diminuiscono i tempi (Figura 4.4) necessari a *development*, *test* e, di conseguenza, alla *release* dell'applicazione:

- 22 giorni necessari alla pianificazione e al design sia dell'applicazione sia dell'infrastruttura in cui si opera
- 10 giorni comprendenti la fase di sviluppo continuo, integrando continuamente nuove funzionalità
- 8 giorni dedicati al testing continuo sulle nuove funzionalità implementate
- 5 giorni per il rilascio in produzione

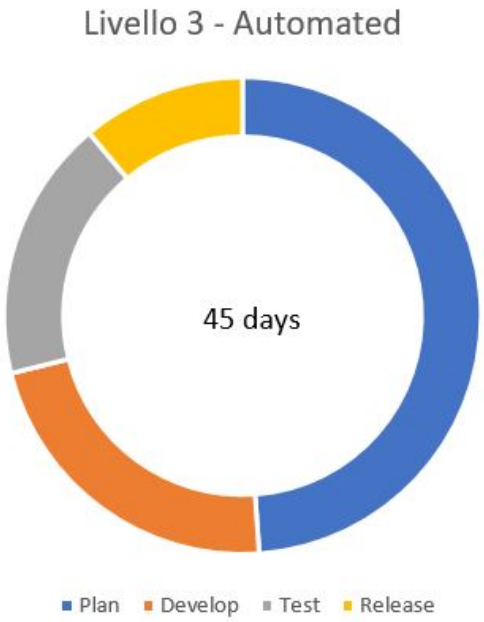

Figura 4.4. Livello *Automated*: le performance

Il *provisioning* definito in fase di pianificazione riduce il grado di flessibilità a questo livello. Circa il 50% del tempo viene impiegato a dover riprogettare, variando uno dei requisiti iniziale, l'intero ambiente in cui effettuare sviluppo e testing. Ad esempio,

nel caso in cui uno dei team abbia bisogno di un *provisioning* specifico per riprodurre un certo tipo di comportamento dell'applicazione, bisognerebbe ritornare al design iniziale e mettere in conto la nuova esigenza. Infine, il rilascio in produzione non è del tutto immediato: seppure con minore probabilità di fallimento, il software continua a ritrovarsi, per la prima volta, in un ambiente completamente nuovo in cui non è stato mai testato precedentemente.

4.3.2 Step 2: da *Automated* a *Continuous*

Automatizzare il processo di *release* è essenziale per la *Continuous Delivery* ma ne costituisce solamente il punto di partenza.

- Adottare un modello flessibile - di tipo *Agile* - capace di adattarsi al volo ai nuovi nuovi requisiti o cambiamenti
- Tracciare qualunque tipo di attività e fornirne un feedback continuo, massimizzando la trasparenza e la collaborazione dei vari team
- Incrementare la frequenza di *deployment*, in maniera da avere, in fase di *release*, una applicazione già pronta ad essere utilizzata in produzione

Rappresentano i tre pilastri di un approccio completo ed efficiente. Ad essi consegue:

- Una risposta più rapida sul mercato (ciclo di *delivery* inferiore a 4 settimane)
- Numero più basso di *incidents* (Riduzione maggiore del 50%)
- Frequenza maggiore di *releases* (Oltre 10 mila per mese)

La *Continuous Delivery* pipeline (Figura 4.5) implementata nel capitolo precedente, rappresenta il punto di controllo dell'intera applicazione. Grazie al suo utilizzo, in pochi minuti è possibile portare a termine un *deployment* completo dell'applicazione. Inoltre, con la riproduzione di ambienti *production-like* e di tool per il *provisioning-on-demand*, si fa in modo che l'applicazione giunga nella fase di *release* già pronta e perfettamente idonea al rilascio in produzione.

Stimando anche qui delle possibili tempistiche (Figura 4.6):

- Nella fase di pianificazione, in questo caso, il focus è più sull'aspetto economico che su quello tecnico. Tralasciando i calcoli economici e ponendo l'attenzione sul design dell'infrastruttura su cui viene eseguita la pipeline, abbiamo avuto modo di vedere come in pochi minuti sia possibile tirare su una infrastruttura virtuale su cui operare

- Considerando che la pipeline, in media, è in grado di effettuare un singolo *deployment*[37] ogni 10 minuti circa, supponiamo che l'applicazione sia pronta - sviluppata e testata - in poco più di una settimana
- Giunta nello stage di *release*, l'applicazione è pronta per il rilascio immediato

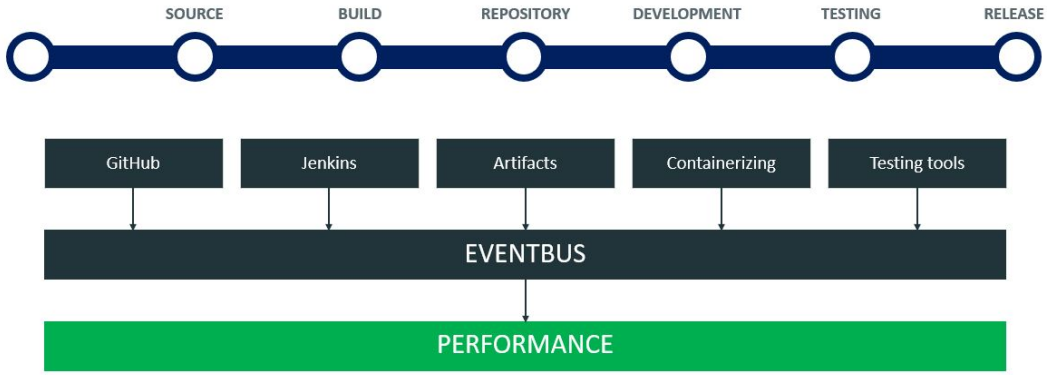

Figura 4.5. *Continuous Delivery*: misurare le performance

A contorno della notevole riduzione dei tempi di *release*, vi è anche un significativo aumento della qualità del software rilasciato (essendo esso continuamente integrato e validato) nonchè una riduzione del rischio di avere una applicazione poco funzionale oppure non corrispondente con quanto richiesto.

4.4 Conclusione e sviluppi futuri (*Third Way*)

La soluzione di *Continuous Delivery* implementata non è affatto un punto di arrivo, ma può essere considerato semplicemente un ottimo punto di transizione per un processo destinato ad ottimizzare ulteriormente la fornitura di software e servizi di qualità.

Dopo aver attraversato la *First Way* e la *Second Way*, l'ulteriore passo in avanti è il passaggio dal livello *Continuous* a quello *Optimized*. Le più grandi organizzazioni sono già in moto verso questo step: ma cosa prevede esso esattamente?

Come già accennato in precedenza, al livello *Optimized* si cerca di conferire alla pipeline di *Continuous Delivery* un certo grado di intelligenza, giungendo a ciò che

[37] Si ricorda che per *deployment* si intende l'intero ciclo di implementazione (integrazione continua del codice), test unitari e test funzionali.

viene definito *zero-touch deployment*. Esso prevede che la pipeline apprenda, in una fase continua di *training*, la serie di problematiche che è possibile incontrare nel *deployment* dell'applicazione. In particolare, si adotta un approccio *fail fast*: l'intento è quello di evitare fallimenti della pipeline in stage avanzati ma (piuttosto) sfruttare i fallimenti iniziali per costruire una storia di informazioni da utilizzare nelle *release* successive. Così facendo, la pipeline è in grado di gestire il *deployment* e i suoi fallimenti, distribuire il *workload* su più *executors* ed eseguire il passaggio in produzione.

Ritornando al presente, con questo lavoro si è voluto dimostrare come, in un mondo in cui l'innovazione tecnologica evolve continuamente, il *DevOps* e la *Continuous Delivery* possano rivelarsi le soluzione ideali. Con l'utilizzo di una serie di strumenti open-source è stato possibile simulare le fasi di esecuzione di una pipeline di rilascio software, capace di ridurre il ciclo di consegna dall'ordine di settimane o mesi a quello dei minuti (Figura 4.6).

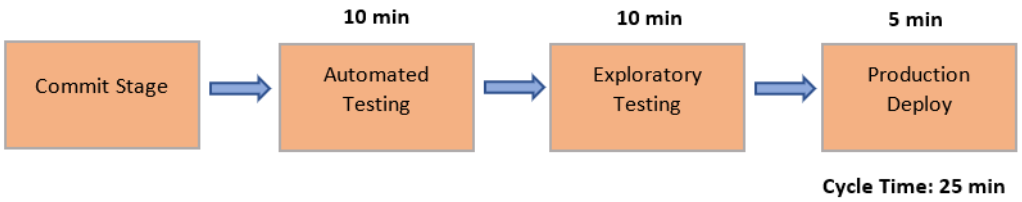

Figura 4.6. Organizzazione performante. Rilascio di una nuova versione stimato nel giro di pochi minuti

L'approccio sistemistico adottato per la realizzazione della pipeline è quello che avviene dietro le quinte di questo nuovo palcoscenico tecnologico. Grandi realtà, quali Amazon e Google, utilizzano questo tipo di implementazione per i loro *Web Services*, fornendo servizi scalabili e di elevata efficienza. Quanto discusso, dunque, ha voluto dare una visione dal basso per comprendere ed esplorare gli ingranaggi di una soluzione che sta cambiando il modo di approcciare lo sviluppo del software, in ogni campo applicativo, con importanti guadagni in termini economici e di prestazioni. Grazie a questo laboratorio è possibile toccare con mano i vantaggi che ne derivano, convincendosi che questa è la strada giusta per adottare una logica dinamica che si adatta in maniera naturale al costante cambiamento del mondo dell'innovazione e della digitalizzazione.

Il segreto del successo è quello di non smettere mai di evolvere.

Appendice A

Continuous Delivery Pipeline: Costruzione mediante *job chain*

A.1 Il Vagrantfile di partenza

https://github.com/marcopu/Job-Chain/tree/master/machines

A.1.1 Configurazione delle machines

- Per il *provisioning* della **JenkinsVM**, fare riferimento allo script **jenkins.sh**

```
config.vm.define "JenkinsVM" do |jenkins|
  jenkins.vm.box_check_update = false
  jenkins.vm.box = "ubuntu/xenial64"
  jenkins.vm.network "forwarded_port", guest: 102, host: 1020
  jenkins.vm.network "private_network", ip: "192.168.90.20"
  jenkins.vm.provision "shell", path: "jenkins.sh"
  jenkins.vm.provider "virtualbox" do |vb1|
    vb1.name = "JenkinsVM"
    vb2.memory = 2048
    vb2.cpus = 2
  end
end
```

- Per il *provisioning* della **AnsibleVM**, fare riferimento allo script **ansible.sh**

```
config.vm.define "AnsibleVM" do |ansible|
  ansible.vm.box_check_update = false
  ansible.vm.box = "ubuntu/xenial64"
  ansible.vm.network "forwarded_port", guest: 101, host: 1010
  ansible.vm.network "private_network", ip: "192.168.90.10"
  ansible.vm.provision "shell", path: "ansible.sh"
  ansible.vm.provider "virtualbox" do |vb2|
    vb2.name = "AnsibleVM"
    vb2.memory = 2048
    vb2.cpus = 2
  end
end
```

- Per il *provisioning* della **DevelopmentVM**, fare riferimento allo script **lamp.sh**

```
config.vm.define "DevelopmentVM" do |development|
  development.vm.box_check_update = false
  development.vm.box = "ubuntu/xenial64"
  development.vm.network "forwarded_port", guest: 103, host: 1030
  development.vm.network "private_network", ip: "192.168.90.30"
  development.vm.provision "shell", path: "lamp.sh"
  development.vm.provider "virtualbox" do |vb3|
    vb3.name = "DevelopmentVM"
    vb3.memory = 2048
    vb3.cpus = 2
  end
end
```

- Per il *provisioning* della **TestingVM**, fare riferimento allo script **sonar.sh**

```
config.vm.define "TestingVM" do |testing|
  testing.vm.box_check_update = false
  testing.vm.box = "ubuntu/xenial64"
  testing.vm.network "forwarded_port", guest: 104, host: 1040
  testing.vm.network "private_network", ip: "192.168.90.40"
  testing.vm.provision "shell", path: "sonar.sh"
  testing.vm.provider "virtualbox" do |vb4|
    vb4.name = "Test6"
    vb4.memory =4096
    vb4.cpus = 2
  end
end
```

A.1.2 Preparazione dell'ambiente

Procedere nel modo seguente:

- Scaricare dal repository specificato - sulla macchina host su cui sono installati **Vagrant** e il provider **VirtualBox** - il contenuto della cartella **machines**
- Digitare - da terminale - il comando **vagrant up** all'interno della cartella **machines**
- Una volta che le macchine saranno avviate e configurate, collegarsi all'indirizzo specificato nel **Vagrantfile** per la JenkinsVM - **http://192.168.90.20:8282** - e seguire la procedura di inizializzazione di Jenkins.

La **JenkinsVM** esegue gli *Ansible playbooks* - tramite protocollo SSH - operando sulla **AnsibleVM**. Nel file **/etc/ansible/hosts** di quest'ultima, aggiungiamo:

```
[AnsibleVM]
192.168.110.10 ansible_ssh_user=root
ansible_python_interpreter=/usr/bin/python3
```

```
[DevelopmentVM]
192.168.110.30 ansible_ssh_user=root
ansible_python_interpreter=/usr/bin/python3

[TestingVM]
192.168.110.40 ansible_ssh_user=root
ansible_python_interpreter=/usr/bin/python3
```

Qualsiasi macchina che si specificherà nel modulo **hosts** dei seguenti *Ansible playbooks*, deve avere un corrispondente identificativo all'interno di questo file.

A.2 I *playbooks* di Ansible

https://github.com/marcopu/Job-Chain/tree/master/playbooks

- **starttomcat.yml**

```
---

- hosts: DevelopmentVM
  remote_user: root
  become: yes
  tasks:

    - name: Check if Apache Tomcat is already installed
      stat: path=/opt/tomcat
      register: check_path

    - name: Installing Tomcat7
      shell:  wget mirror.nohup.it/apache/tomcat/tomcat-7/v7.0.82/bin/apache-tomcat-7.0.82.tar.gz;
              tar xvzf apache-tomcat-7.0.82.tar.gz;
              mv apache-tomcat-7.0.82 /opt/tomcat;
              cd /root;
              sudo echo 'export JAVA_HOME=/usr/lib/jvm/java-8-oracle' >> .bashrc;
              echo 'export CATALINA_HOME=/opt/tomcat' >> .bashrc;
      when: check_path.stat.exists == false

    - name: Starting Tomcat7
      command: nohup /opt/tomcat/bin/startup.sh
```

Il job *Orchestrate* - tramite SSH - esegue il comando:

```
ansible-playbook /playbook/starttomcat.yml
```

Il *playbook* viene eseguito sulla macchina specificata nel modulo **hosts**, ossia la **DevelopmentVM**.

- **unit.yml**

```
---

- hosts: AnsibleVM
  remote_user: root
  become: yes
  tasks:

    - name: Unit Testing
      shell: cd /home/app; mvn test
```

- **analyzer.yml**

```
---

- hosts: AnsibleVM
  remote_user: root
  become: yes
  tasks:

    - name: Code Analysis
      shell: cd /home/app; mvn sonar:sonar
```

I *playbooks* **unit.yml** e **analyzer.yml** vengono eseguiti in locale sulla **AnsibleVM**, rispettivamente dai job *UnitTesting* e *CodeAnalysis* creati in Jenkins.

Gli *artifacts* trasferiti dalla **JenkinsVM** vengono copiati nella cartella **/home** della **AnsibleVM**. Grazie alle dipendenze definite nel file di configurazione **pom.xml**, i comandi **maven** eseguono rispettivamente test unitari e analisi del codice.

- **functional.yml**

```
---

- hosts: TestingVM
  remote_user: root
  sudo: yes
  tasks:

    - name: Create folder for test reports
      file: path=/reports state=directory

    - name: Testing on Mozilla and Chrome browsers
      shell: python /home/selenium/suite.py  >> /reports/"report_$(date +"%Y_%m_%d_%I_%M").html"
```

Eseguito dal job *FunctionalTesting*, questo *playbook* esegue sulla **TestingVM** i *functional tests* contenuti nello script Python **suite.py**

A.3 L'applicazione *app*

https://github.com/marcopu/Job-Chain/tree/master/app

All'interno della cartella **app** del repository viene riportato il file di configurazione **pom.xml** con tutte le sue dipendenze e proprietà di configurazione. Ad esempio:

- Per Maven

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
</plugin>
```

- Per SonarQube

```
<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <sonar.host.url>http://192.168.90.40:9000/sonar</sonar.host.url>
</properties>
...
<plugin>
  <groupId>org.sonarsource.scanner.maven</groupId>
  <artifactId>sonar-maven-plugin</artifactId>
  <version>3.3.0.603</version>
</plugin>
```

- Per il framework JUnit

```
<dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.5</version>
   <scope>test</scope>
</dependency>
```

A.3.1 Code Analysis - SonarQube

Per l'utilizzo di SonarQube, modificare il file **/opt/sonar/conf/sonar.properties** - sulla **TestingVM** - nel seguente modo:

```
sonar.jdbc.username=sonar
sonar.jdbc.password=sonar
sonar.jdbc.url=jdbc:mysql://192.168.90.30:3306/sonar?
useUnicode=true&characterEncoding=utf8&rewriteBatchedStatements=true
&useConfigs=maxPerformance

sonar.web.host=192.168.90.40
sonar.web.context=/sonar
sonar.web.port=9000
```

L'indirizzo `192.168.90.40:9000` corrsiponde all'host SonarQube configurato.

L'indirizzo `192.168.90.30:3306` corrisponde al MySQL Server - requisito di funzionamento per SonarQube - installato sulla **DevelopmentVM**.

A.4 Fase di *testing*

A.4.1 Unit tests - JUnit

Seguendo un approccio *test-driven development*, si definiscono le classi di test:

- **`TicketTest.java`** contenente il caso di test nella forma:

```
@Test
public void test1() {
Ticket instance = new Ticket();
String customer = "Consulente Pippo";
instance.setCustomerName(customer);
assertEquals(instance.getCustomerName(), customer);
}

@Test
public void test2() {
Ticket instance = new Ticket();
instance.setCustomerName("Consulente Paperino");
String customer = instance.getCustomerName();
assertEquals(customer, "Cosulente Paperino");
}
```

- **`AttachmentTest.java`** contenente il caso di test nella forma:

```
@Test
public void test3() {
Attachment instance = new Attachment();
String name = "Allegato";
instance.setName(name);
assertEquals(instance.getName(), name);
}
```

Si tratta di test elementari eseguiti da chi sviluppa per controllare il corretto comportamento dei metodi definiti

A.4.2 Functional Acceptance tests - Selenium

Sulla **TestingVM**, vengono installati da terminale i moduli Python e Selenium:

```
apt-get install python-pip
pip install selenium
```

Per scaricare i driver compatibili ai browser con cui interfacciarsi per effettuare i test funzionali:

- **chromedriver** per Google Chrome
  https://sites.google.com/a/chromium.org/chromedriver/home
- **geckodriver** per Mozilla Firefox
  https://github.com/mozilla/geckodriver/releases

I casi di test compongo la **suite.py** contenuta nella cartella **selenium** del repository di riferimento **https://github.com/marcopu/Job-Chain**

- Sul browser Mozilla Firefox:

```
class MozillaTesting(unittest.TestCase):

    display = Display(visible=0, size=(1366, 768))
    display.start()

    def setUp(page):
        page.driver = webdriver.Firefox()

    def test_openMozilla(page):
        driver = page.driver
        driver.get("http://192.168.90.40:8080/app/tickets")
        page.assertIn("Consoft", driver.title)

    def test_insertTicketMozilla(page):
        driver = page.driver
        driver.get("http://192.168.90.40:8080/app/tickets")
        inputElement = page.driver.find_element_by_link_text("Crea il tuo Ticket")
        inputElement.click()

    def tearDown(page):
        page.driver.close()
```

- Sul browser Google Chrome:

```
class ChromeTesting(unittest.TestCase):

    def setUp(page):
        options = webdriver.ChromeOptions()
        options.add_argument('headless')
        options.add_argument('window-size=1366x768')
        options.add_argument('no-sandbox')
        page.driver = webdriver.Chrome(chrome_options=options)

    def test_openChrome(page):
        driver = page.driver
        driver.get("http://192.168.90.40:8080/app/tickets")
        page.assertIn("Consoft", driver.title)

    def test_insertTicketChrome(page):
        driver = page.driver
        driver.get("http://192.168.90.40:8080/app/tickets")
        inputElement = page.driver.find_element_by_link_text("Crea il tuo Ticket")
        inputElement.click()

    def tearDown(page):
        page.driver.close()
```

All'interno della medesima cartella, è contenuto lo script **`HTMLTestRunner.py`** tramite cui vengono generati i report HTML. Essi vengono salvati nella cartella **`/reports`** - creata appositamente dal playbook **`functionaL.yml`** - sulla **`TestingVM`**.

Appendice B

Continuous Delivery Pipeline: Implementazione via *script*

B.1 Il Vagrantfile *master*

https://github.com/marcopu/Declarative-Pipeline/tree/master/vagrant%20master

```
#sintassi ruby

Vagrant.configure("2") do |config|
   config.vm.provision "shell", inline: "echo starting Jenkins Master"
   config.vm.define "JenkinsVM" do |jenkins|
      jenkins.vm.box_check_update = false
      jenkins.vm.box = "ubuntu/xenial64"
      jenkins.vm.network "forwarded_port", guest: 80, host: 8690
      jenkins.vm.network "private_network", ip: "192.168.110.20"
      jenkins.vm.provision "shell", path: "jenkins.sh"
      jenkins.vm.provider "virtualbox" do |vb1|
        vb1.name = "JenkinsVM"
        vb1.memory = 2048
        vb1.cpus = 2
      end
    end

  config.trigger.after :up do
     system('./myscript.bat')
  end

  config.trigger.after :halt do
     system('./stopslave.bat')
  end
end
```

La **JenkinsVM** viene avviata e ne viene effettuato il *provisioning* tramite lo script shell **jenkins.sh**.

Con il comando **vagrant up** viene inoltre eseguito - sulla macchina host Windows - il file batch **myscript.bat** che instaura la comunicazione tra il nodo *master* -

JenkinsVM - e il nodo *slave* **winslave** sull'host Windows stesso. Il file batch avvia il JAR **agent.jar** scaricabile dal repository specificato.

Quando verrà eseguito il comando **vagrant halt** al termine della pipeline, verrà eseguito il file batch **stopslave.bat** per arrestare il nodo **winslave** sull'host Windows.

Grazie allo *slave* configurato, la **JenkinsVM** sarà in grado di gestire le macchine configurate nel **Vagrantfile** contenuto nella cartella **vagrant slaves**.

B.2 Il Vagrantfile *slaves*

https://github.com/marcopu/Declarative-Pipeline/tree/master/vagrant%20slaves

In questo file vengono definite le macchine che il nodo **winslave** - sotto delega della **JenkinsVM** - è incaricato di avviare tramite un **vagrant up** sull'host Windows.

Configurazioni e script sono consultabili nel repository specificato.

B.3 Utilizzo di Docker

https://github.com/marcopu/Declarative-Pipeline

- Riproduzione di un ambiente *production-like* mediante l'utilizzo del *playbook* **dockerplay.yml** - riportato nella cartella **playbooks** del repository - il quale utilizza il **Dockerfile**

  ```
  FROM tomcat
  ADD app.war /usr/local/tomcat/webapps/
  CMD ["catalina.sh", "run"]
  ```

- Reset delle configurazioni al termine dell'esecuzione della pipeline mediante l'utilizzo del *playbook* **dockerclean.yml** - riportato nella cartella **playbooks** del repository specificato.

B.4 Implementazione della Declarative Pipeline

https://github.com/marcopu/Declarative-Pipeline/blob/master/Jenkinsfile

- Dalla cartella **master**, sull'host Windows, viene avviata la **JenkinsVM** - con il comando **vagrant up** - e successivamente il nodo **winslave** tramite script batch.
- La **JenkinsVM** delega al nodo **winslave** l'avvio delle macchine definite all'interno del **Vagrantfile**, contenuto nella cartella **slaves**, sull'host Windows.

```
stage("Starting slaves from Windows host"){

   agent{node 'winslave'}

   steps{
     bat 'cd C:/Users/Administrator/Desktop/slaves & vagrant up'
   }
}
```

- Stage di compilazione e *packaging* dell'applicazione[38].

```
stage("Creating archive WAR"){
   steps{
     sh 'mvn -f ./app/pom.xml clean -DskipTests'
     sh 'mvn -f ./app/pom.xml compile -DskipTests'
     sh 'mvn -f ./app/pom.xml package -DskipTests'
   }
}
```

- Stage di esecuzione degli *unit tests* tramite **unit.yml**

```
stage("Unit testing"){
   steps{
     sh 'ssh root@192.168.110.10 mkdir -p /pipeline'
     sh 'scp -r ** root@192.168.110.10:/pipeline'
     sh 'ssh root@192.168.110.10 ansible-playbook /pipeline/playbooks/unit.yml'
   }
}
```

[38]Si osservi come non venga specificato il salvataggio degli *artifacts*. In *Jenkins* viene creato un solo job di tipo *Pipeline*, tutti gli *artifacts* vengono così condivisi - tra i vari stage - tramite il *workspace* del job comune.

- Stage di analisi del codice tramite **`analyzer.yml`**

```
stage("Code Analysis"){
   steps{
     sh 'ssh root@192.168.110.10 ansible-playbook /pipeline/playbooks/analyzer.yml'
   }
}
```

- Installazione di *Apache Tomcat* - tramite **`starttomcat.yml`** - ed esecuzione dell'applicazione sulla **`DevelopmentVM`**.

```
stage("Development"){
   steps{
     sh 'ssh root@192.168.110.10 ansible-playbook /pipeline/playbooks/starttomcat.yml'
     sh 'ssh root@192.168.110.30 rm -f /opt/tomcat/webapps/*.war'
     sh 'scp -r app/target/*.war root@192.168.110.30:/opt/tomcat/webapps'
   }
}
```

- Creazione - da **`Dockerfile`** - di un ambiente *production-like* in cui eseguire il testing *automated* dell'applicazione.

```
stage("Building environment for Automated Testing"){
   steps{
     sh 'ssh root@192.168.110.50 mkdir -p /dockerfolder'
     sh 'scp Dockerfile root@192.168.110.50:/dockerfolder'
     sh 'scp -r app/target/*.war root@192.168.110.50:/dockerfolder'
     sh 'ssh root@192.168.110.10 ansible-playbook /pipeline/playbooks/dockerplay.yml'
   }
}
```

- Stage di esecuzione dei *functional tests* tramite **`functional.yml`**

  Come *post actions* di questo stage, l'ambiente viene resettato - sia con semplici comandi di rimozione sia con il *playbook* **`dockerclean.yml`** - in maniera da non impattare sulle esecuzioni successive della pipeline.

```
stage("Functional Testing"){
   steps{
     sh 'ssh root@192.168.110.40 mkdir -p /pipeline'
     sh 'scp -r ** root@192.168.110.40:/pipeline'
     sh 'ssh root@192.168.110.10 ansible-playbook /pipeline/playbooks/functional.yml'
   }

   post{
     ...
     echo 'Deleting workspace'
     deleteDir()

     echo 'Cleaning. . .'
     sh 'ssh root@192.168.110.10 ansible-playbook /pipeline/playbooks/dockerclean.yml'
     sh 'ssh root@192.168.110.10 rm -r /pipeline'
     sh 'ssh root@192.168.110.40 rm -r /pipeline'
     sh 'ssh root@192.168.110.50 rm -r /dockerfolder'
   }
}
```

- La **JenkinsVM** delega al nodo **winslave** il compito di sospendere le macchine.

```
stage("Suspending machines"){
   agent{node 'winslave'}

   steps{
     bat 'cd C:/Users/Administrator/Desktop/slaves & vagrant suspend'
   }
   ...
}
```

Bibliografia

[1] Bell, T.E. et T.A. Thayer, 'Software Requirements: Are They Really A Problem?', *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco (California, USA), TRW Defense and Space Systems Group, 1976, pp. 61-68

[2] Royce, W.W., 'Managing the Development of Large Software Systems: concepts and techniques', *Proceedings of the 9th International Conference on Software Engineering*, Monterey (California, USA), TRW Software Series, 1970, pp. 328-338

[3] Parnas, D.L. et P.C. Clements, 'A Rational Design Process: How and Why to Fake It', *IEEE Transactions on Software Engineering*, Volume 12, NO. 2, 1986, pp. 251-257

[4] The Agile Alliance, *Manifesto for Agile Software Development*, 2001, http://agilemanifesto.org, (accessed 14 Febbraio 2018)

[5] Cockcroft, A., 'Velocity and Volume (or Speed Wins)', (Flowcon, Dicembre 2013), https://www.youtube.com/watch?v=wyWI3gLpB8o, (accessed 15 Febbraio 2018)

[6] Brown, A. et al., 'State Of DevOps Report', 2016, Puppet Labs and DORA, https://puppet.com/resources/whitepaper/2016-state-of-devops-report, (accessed 14 Febbraio 2018)

[7] Null, C., *How to Measure DevOps ROI*, https://techbeacon.com/devops-roi-how-measure-guide (accessed 14 Febbraio 2018)

[8] Humble, J. et D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, USA, Addison-Wesley Signature Series, 2011

[9] Ravichandran, A. et al., *DevOps for Digital Leaders*, New York (USA) , CA Press, 2016

[10] Kim, G. et al., *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*, Portland (USA), IT Revolution, 2016

[11] North, D., *Introducing BDD*, 2006, https://dannorth.net/introducing-bdd (accessed 14 Febbraio 2018)