UNIVERSITÀ DEGLI STUDI DI MILANO - BICOCCA
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

DevOps: studio e implementazione di una pipeline di CI e CD nel progetto Sphere

Relatore: Prof. Mariani Leonardo

Tutor Aziendale: Dott. Mesiano Cristian

Relazione della prova finale di:
Renzo Simone
Matricola 781616

Anno Accademico 2019-2020

Abstract

In un mondo in continuo sviluppo, la necessità di adattarsi velocemente al cambiamento è spesso ciò che permette di contraddistinguere realtà di successo dalle fallimentari. Il software è probabilmente uno dei prodotti che più segue questa filosofia di cambiamento repentino, con la continua uscita di nuove tecnologie, nuove metodiche e la conseguente necessità di cambiare spesso rotta e requisiti in base alle necessità o al mercato di riferimento.

La nascita di metodi *Agile* e di nuove filosofie improntate all'unire ciò che prima era separato, in un unico processo, hanno permesso di adattarsi con successo ai cambiamenti, rendendo l'industria del software quella più all'avanguardia e resiliente nel tempo, continuando tutt'oggi a migliorarsi sempre più.

> *It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change.*
>
> Charles Darwin,

Indice

Capitolo 1

Introduzione

1.1 Il Contesto Aziendale

1.1.1 Storia

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

1.1.2 Il progetto Sphere

In un mondo dove l'informazione risiede principalmente sotto forma di immagini, Sphere si propone come soluzione per trasformare il fisico in digitale, rompendo la barriera che divide una semplice fotografia da un elemento tangibile.

Figura 1.1: Sphere Logo

Grazie a Sphere l'utente è in grado di creare un *Digital Inventory* di casa propria, partendo da delle semplici fotografie panoramiche acquisite da lui stesso analizzate e convertite poi in oggetti reali grazie ai modelli di AI che permettono di riconoscere oggetti, materiali e ambienti in modo dinamico. Questi dati possono essere sfruttati specialmente a fini di *risk-management* in ambito assicurativo, velocizzando il processo di **digital claim management**, quoting e premio

assicurativo, attraverso un indice dinamico di "bontà" dell'utenza chiamato *Sphere Index*.

Sphere non è un prodotto unico ma un prodotto **modulare**, ogni cliente può quindi personalizzare la propria esperienza integrando i moduli necessari al proprio business, senza preoccuparsi di dipendenze esterne o di costi non controllabili.

1.1.3 Profilo personale in azienda

In PerceptoLab Srl, il candidato è stato assunto in Giugno 2020, con ruolo di Infrastructure & DevOps Engineer, integrato nel team di Backend Development.

1.2 Scopo del Project Work

1.2.1 Obiettivi

Il Project Work ha come scopo il design e l'implementazione di un processo che includa una Pipeline di Continuous Integration e di Continuous Delivery, nell'ambito del progetto Sphere.

In particolare si prefigge questi obiettivi:

1. Creazione di un processo di *Continuous Integration* per il repository progettuale, mediante l'uso di tool per il testing automatico e per la gestione delle Pull Request, con integrazione per build in ambiente macOS;
2. Creazione di un processo di *Continuous Delivery* per la creazione di immagini Docker mediante utilizzo di tag specifici su repository e delivery degli artefatti su registry remoto;
3. Integrazione nella pipeline DevOps di *analisi statica e dinamica* del codice mediante tools dedicati e definizione di quality gates in base alle necessità progettuali;
4. Creazione e gestione dell'infrastruttura (basata su *Amazon Web Services*) necessaria al deployment dei servizi sviluppati nel progetto aziendale.

1.2.2 Pianificazione del Lavoro

Il Project Work si è svolto durante il periodo di 3 mesi tra l'1 Ottobre 2020 ed il 31 Dicembre 2020, in modalità di remote working con l'utilizzo di tools di collaboration integrati in Google GSuite (Google Chat, Meets) e nella suite Atlassian (Bitbucket, Jira, Confluence).

1.2.3 Prodotti Finali

I prodotti del project work saranno i seguenti:

- Analisi dei Requisiti per i processi da implementare
- Pipeline di Continuous Integration per i servizi di Backend e Mobile
- Pipeline di Continuous Delivery per i servizi di Backend (Docker Containers)
- Quality Assurance Gates basati sulla analisi dei test e del codice con tools dedicati
- Infrastruttura basata su Amazon Web Services (Risorse Cloud, VMs) per gestire i processi descritti.

Capitolo 2

Modelli di Sviluppo del Software

2.1 Il ciclo di vita del software

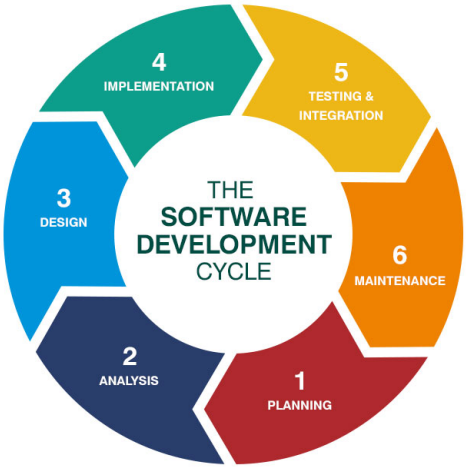

Figura 2.1: Software Development Lifecycle

2.1.1 Le fasi del ciclo di vita

Il ciclo di vita del software si riferisce ad una metodologia che permetta di ottenere software di massima qualità al minimo costo di produzione e nel minor tempo possibile, dividendo la sua vita in diverse fasi consequenziali:

1. Planning e Analisi
2. Design
3. Implementazione
4. Testing e Integrazione
5. Manutenzione

Planning e Analisi Si analizzano i sistemi esistenti per i cambiamenti necessari ed il problema da risolvere in termini di software development. Questa fase crea in output una serie di *Requisiti* che possono essere Funzionali, Non Funzionali, o di Dominio, ed un piano di lavoro per sviluppare tali requisiti in un tempo definito (ma, come vedremo, variabile in metodi *Agile*). Le definizioni di tali requisiti sono dettate dallo standard *IEEE 610.12-1990*.

Design I requisiti vengono trasformati in una *specifica di Design* (architetturale ed implementativa), che verrà in seguito analizzata dagli *stakeholders*, ottenendo così feedback e suggerimenti in base alle esigenze. In questa fase diventa cruciale implementare un sistema per incorporare i feedback così da migliorare il design finale ed evitare costi aggiuntivi a fine sviluppo.

Implementazione Questa fase inizia lo sviluppo del software in se, seguendo la specifica di design della fase precedente, ed utilizzando convenzioni, code style, pratiche e linee guida comuni per tutti i soggetti coinvolti nello sviluppo. L'utilizzo di linee guida comuni permette di evitare fraintendimenti all'interno del team di sviluppo, e di facilitare le fasi future di manutenzione.

Testing e Integrazione Il software sviluppato viene sottoposto a test per difetti e mancanze, risolvendo i problemi trovati lungo il percorso e migliorando le feature implementate fino ad arrivare ad una qualità in linea con le specifiche originali. In seguito, viene integrato con il resto dell'ambiente mediante deployment, così da poterlo iniziare ad utilizzare in casi reali.

Manutenzione Alla fine del processo, difficilmente si saranno raggiunti tutti i requisiti alla perfezione, motivo per cui la fase di manutenzione gioca un ruolo fondamentale per gestire tutto ciò che segue lo sviluppo principale del software. Questa fase permette quindi di analizzare i comportamenti sul campo del software sviluppato, così da agire di conseguenza nel risolvere problemi in modo più mirato.

2.1.2 I modelli classici: Waterfall e Iterativo

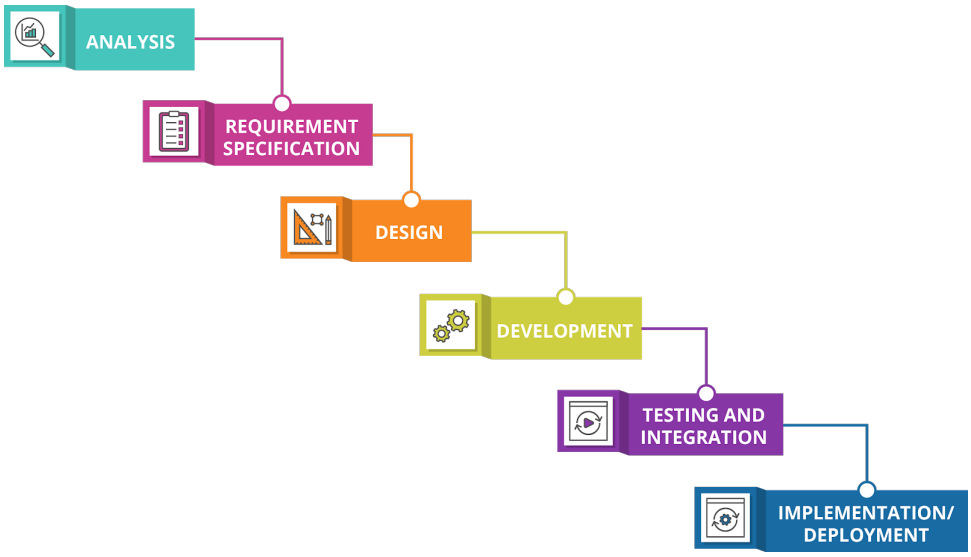

Figura 2.2: Modello Waterfall

Waterfall Il processo di sviluppo storicamente più tradizionale e semplice è chiamato *Waterfall*. Il nome suggerisce come, rispetto alle fasi del ciclo di vita del software, queste vengano eseguite in "cascata", dove la fine di una fase permette di iniziare quella successiva, seguendo ciò che era stato appreso dalla produzione manufatturiera applicandolo in ambito dello sviluppo software.

La creazione di tale processo ha permesso di superare i limiti del processo *code and fix*, permettendo di pianificare in modo più strutturato e dividendo in modo netto le problematiche in base alla fase di appartenenza. Altrettante sono però state le problematiche derivanti dalla sua applicazione, tra cui:

- Le fasi di *alpha/beta* testing ripercorrono per natura tutte le fasi del processo, rallentando lo sviluppo;
- Ogni fase viene congelata dopo la sua fine, rendendo impossibile la comunicazione tra clienti e sviluppatori dopo la fase iniziale;
- La pianificazione viene effettuata solo all'inizio, orientando lo sviluppo ad una data specifica di rilascio; Ogni errore porta a ritardare tale data, che non può però essere stimata di nuovo;
- La stima dei costi e delle risorse si rende difficile senza la prima fase di Analisi;
- La specifica di requisiti vincola il prodotto da sviluppare, mentre nei casi reali spesso le necessità del cliente cambiano in corso d'opera, specialmente sul lungo termine;

Figura 2.3: Modello Iterativo

Iterativo Una evoluzione del processo *Waterfall* è il modello *Iterativo*, basato sullo stesso ciclo di vita descritto precedentemente, ma con un sostanziale cambiamento che lo rende la base dei modelli odierni, ovvero il riconoscimento che lo sviluppo di un software non è composto di una singola iterazione del flusso ma di diverse iterazioni, sempre incrementali.

L'idea alla base del modello *Iterativo* consta nel ripetere il ciclo di sviluppo più volte, in porzioni di tempo più ristrette, permettendo agli sviluppatori di apprendere dai cicli precedenti e di migliorare i successivi, grazie alla continua revisione dei requisiti e del design del software.

Il processo parte con una prima iterazione volta a creare un prodotto basilare ma usabile, in modo da raccogliere il feedback dell'utente o cliente da utilizzare come input per il ciclo

successivo. Per guidare le varie iterazioni, si sfrutta una lista di tasks necessari per lo sviluppo del software, che include sia nuove feature sia modifiche al design provenienti da iterazioni precedenti, da aggiornare in ogni fase di analisi (per ogni iterazione).

Confrontato con *Waterfall*, il modello *Iterativo* porta diversi vantaggi:

- L'utente viene coinvolto ad ogni iterazione, migliorando il feedback e la qualità del prodotto finale;
- Ogni iterazione incrementale produce un *deliverable* che può essere accettato dall'utente, e solo dopo ciò si potrà procedere alla prossima iterazione;
- Ogni iterazione permette di rimodulare le risorse necessarie allo sviluppo, così da attuare tecniche di cost-saving;
- Il prodotto può essere consegnato fin dalla prima iterazione, seppur in fase embrionale ma funzionante;
- Il modello *Iterativo* può essere applicato anche a progetti di piccole dimensioni con successo.

2.2 Modelli *Agile*

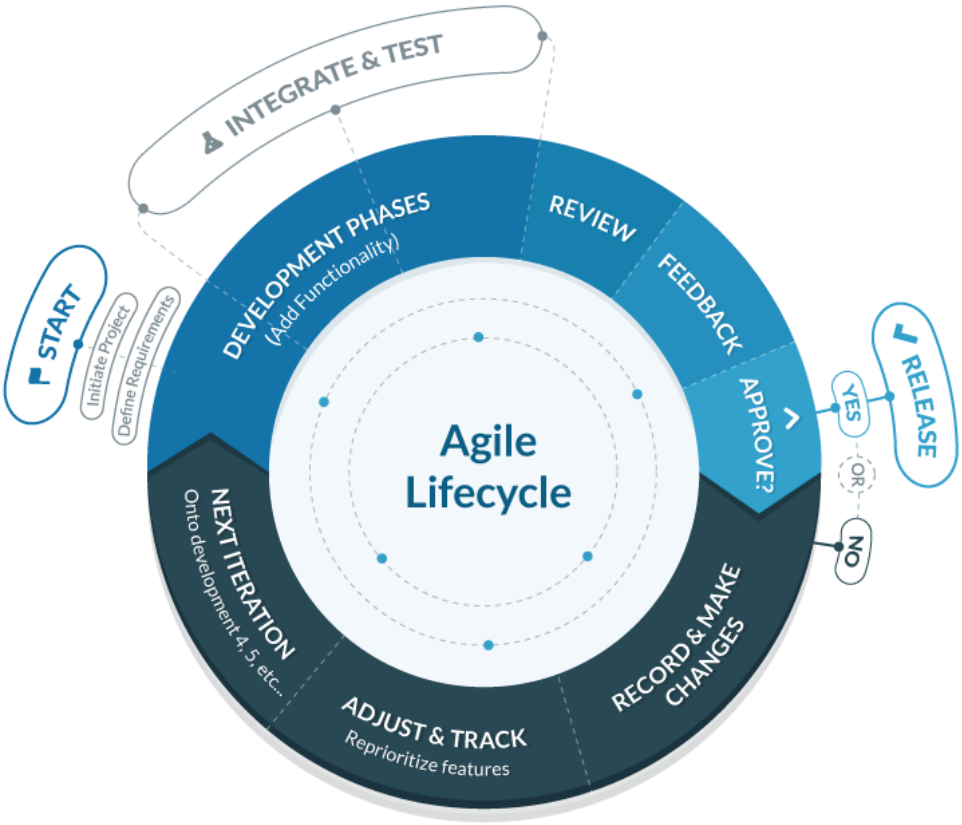

Figura 2.4: Modello Agile Generico

2.2.1 La filosofia *Agile*

La filosofia *Agile* è emersa a partire dai primi anni 2000, grazie alla pubblicazione del *"Manifesto for Agile Software Development"* nel 2001, nato dall'evoluzione dei metodi classici iterativi ed incrementali.

L'utilizzo di questa filosofia permette di ridurre sensibilmente il rischio di errori dovuti alla male interpretazione dei requisiti o il ritardo nelle tempistiche di consegna del prodotto finale, ponendo il focus sulla **adattabilità** dei processi al cambiamento e sulla **soddisfazione** del cliente come metrica di successo di un processo di sviluppo. Ciò è permesso grazie alla suddivisione dello sviluppo in **iterazioni** di piccole dimensioni (generalmente 1-3 settimane), permettendo un rilascio continuo del software in modo incrementale, e dall'attuazione di alcuni principi tra cui:

- **Individui e interazioni**: nello sviluppo *Agile*, auto-organizzazione e motivazione sono importanti tanto quanto le interazioni tra persone e il pair programming;
- **Software Funzionante**: viene consegnato frequentemente software funzionante in tempi preferibilmente brevi, sotto forma di *Demo*;
- **Collaborazione col Cliente**: committenti a sviluppatori devono lavorare insieme quotidianamente per tutta la durata del progetto;
- **Risposta al Cambiamento**: lo sviluppo *Agile* si concentra nel fornire una risposta veloce ai cambiamenti nei requisiti, anche in fasi avanzate dello sviluppo;
- **Gestione delle Priorità**: lo sviluppo inizia solo dopo aver messo in priorità gli obiettivi, spesso utilizzando una tecnica chiamata *MoSCoW* (*Must - Should - Could - Won't Have*);
- **Timeboxing**: suddividere il progetto di sviluppo in intervalli temporali ben definiti, spesso di pochi giorni o settimane, entro il quale consegnare alcune features, contenuti in intervalli più lunghi di consegna del prodotto finale o di una parte di esso.

I modelli *Agile* presentano diversi vantaggi che han permesso la loro adozione nella maggior parte degli ambienti di sviluppo software, tra cui:

- I processi sono molto **realistici** e riflettono il mercato;
- Promuovono il lavoro in team e il cross-training;
- Le funzionalità vengono sviluppate velocemente e dimostrate al cliente;
- Utilizzabili sia con requisiti fissati che in continuo cambiamento;
- Pianificazione ridotta al minimo;
- Facilità di gestione.

Contemporaneamente portano anche alcuni svantaggi, derivanti dalla flessibilità del processo e quindi dalla non applicabilità in tutti i casi esistenti:

- Difficoltà di gestione di dipendenze complesse;
- Rischio aumentato sulla sostenibilità e mantenibilità del processo;
- Necessità di un piano di lavoro generale, e di **figure chiave** per far funzionare il processo;
- Forte dipendenza dal feedback del cliente, per colpa di cui il team potrebbe essere direzionato in modo errato.

2.2.2 Un caso di successo: SCRUM Framework

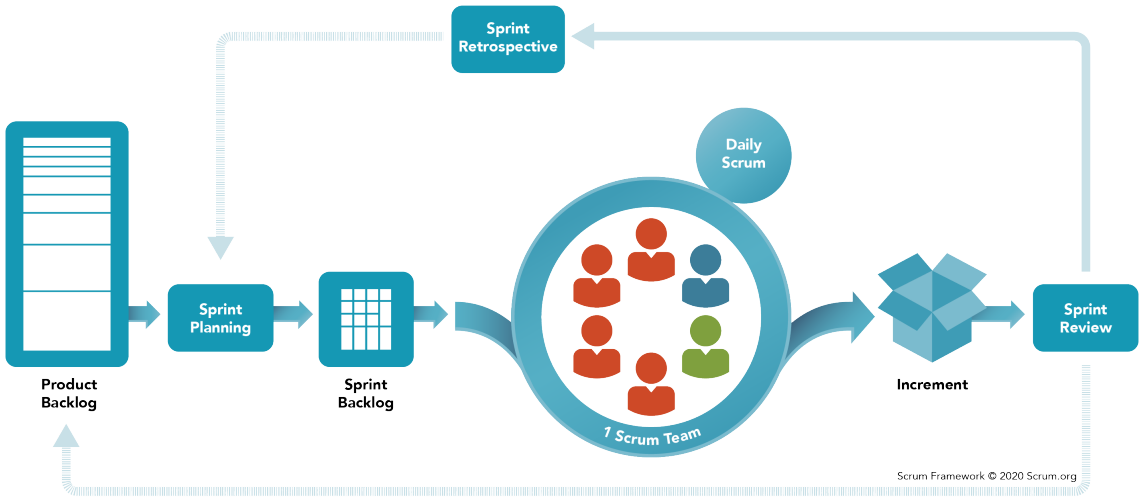

Figura 2.5: SCRUM Framework

Nel mondo *Agile* sono nati diversi metodi e processi con approcci differenti alla filosofia precedentemente descritta, uno dei quali è **SCRUM Framework**, il più diffuso ed utilizzato nel mondo dello sviluppo software.

Il termine **SCRUM** deriva dal gioco del *Rugby*, più precisamente dal pacchetto di mischia, e viene usato come metafora del team di sviluppo che deve lavorare insieme in modo che tutti gli attori del progetto *spingano* nella stessa direzione, agendo come una unica entità coordinata.

Notiamo come non stiamo definendo un processo, ma bensì un *framework*, e come tale si propone come collezione di metodologie che sposano la filosofia *Agile* ma che possono essere implementate al meglio delle proprie possibilità, anche con minimi adattamenti.

SCRUM si basa sulla teoria dei controlli empirici di analisi strumentale e funzionale di processo, e si basa su alcuni pilastri derivanti da tale teoria:

1. **Trasparenza**: gli aspetti rilevanti del processo devono essere visibili ai responsabili. Questa trasparenza richiede che venga utilizzato un *linguaggio comune* condiviso da tutti, ed una definizione di *done* (finito) condivisa tra gli addetti ai lavori e chi deve accettarlo;

2. **Ispezione**: l'uso di SCRUM prevede l'ispezione frequente degli *artefatti* prodotti ed i progressi realizzati verso gli obiettivi stabiliti, individuando precocemente eventuali deviazioni. La frequenza di tali ispezioni deve essere tale da non rallentare il corso dei lavori, e deve essere effettuato da ispettori qualificati;

3. **Adattamento**: se durante l'ispezione si trovano difformità oltre i limiti accettabili, bisogna intervenire sul processo stesso e sul materiale prodotto. L'intervento deve essere portato a termine il più velocemente possibile per evitare ulteriori ripercussioni e ridurre al minimo le perdite. Vengono quindi definite quattro occasioni per ispezione e adattamento:

    - Sprint Planning Meeting;

- Daily Scrum;
- Sprint Review;
- Sprint Retrospective.

.

Il Team SCRUM In un processo dove si utilizza *SCRUM*, viene definito il team di sviluppo come *Team SCRUM*, formato da diverse figure:

- **Product Owner**: rappresenta gli stakeholders (cliente), ed è responsabile del valore business del team. Il PO definisce gli *item* (requisiti di prodotto) in base ai bisogni dei clienti (usando *user stories*), assegnando la loro priorità e li inserisce nel *product backlog*;
- **Team di Sviluppo**: l'insieme degli sviluppatori responsabili della consegna del prodotto, con incrementi potenzialmente rilasciabili alla fine di ogni **Sprint**. Il team è composto da un numero di persone che varia da 3 a 9, con competenze cross-funzionali e che si auto-organizza;
- **Scrum Master**: responsabile della rimozione di ostacoli che potrebbero limitare la capacità produttiva del team e quindi di raggiungere gli obiettivi dello *Sprint*. Sebbene possa sembrare un ruolo manageriale, lo Scrum Master è solo il supervisore del processo, detta l'autorità relativa alla applicazione delle norme e presiede le riunioni importanti, e funge da protezione al team di sviluppo che può così concentrarsi sullo sviluppo.

Sprint Nello sviluppo *SCRUM*, l'unità base di tempo è chiamato *Sprint*, generalmente di durata di 1-4 settimane. Ogni Sprint inizia con una **riunione di pianificazione**, e si conclude con una **riunione di revisione** del raggiungimento degli obiettivi. Durante lo Sprint non è inoltre permesso cambiare gli obiettivi prefissati all'inizio, che verranno quindi tenute in considerazione nell'iterazione successiva.

Al termine di ogni Sprint, il team consegna una versione potenzialmente completa e funzionale del prodotto, contenente gli sviluppi conclusi nello Sprint stesso e quelli dei precedenti già conclusi. Le funzionalità da sviluppare in uno Sprint provengono dal *product backlog*, come compilato e prioritizzato dal *Product Owner*, ed una volta inserite nello **sprint backlog** non possono essere più aggiunte ne rimosse durante il corso dell'iterazione.

Eventi In SCRUM gli eventi vengono sfruttati per **creare una routine** e ridurre al minimo riunioni al di fuori di quelle definite dal framework stesso, tali eventi sono inoltre inclusi nel timeboxing del processo così da integrarsi al meglio con lo sviluppo stesso senza bloccare il team più del dovuto.

Sprint Planning All'inizio di ogni Sprint, viene effettuato un meeting volto a pianificare le attività dello stesso e gli obiettivi da conseguire entro la sua conclusione, e coinvolge tutto lo Scrum Team. Al suo interno viene definito lo *sprint backlog* e si ottiene una stima delle tempistiche e risorse necessarie al suo completamento.

Daily Scrum Ogni giorno durante lo Sprint, viene effettuata una riunione di comunicazione con tutto il team, in cui ogni componente aggiorna gli altri con la sua situazione di sviluppo. Tale meeting viene effettuato nello stesso posto e ora ogni giorno, in un tempo ben definito e ridotto (15 minuti massimo), e viene generalmente effettuato in piedi (da qui il nome *daily standup*).

Sprint Review Alla fine dello Sprint viene effettuato un meeting volto a ispezionare l'incremento e adattare, se necessario, il *product backlog*. Durante tale riunione il team di sviluppo e gli *stakeholders* collaborano su ciò che è stato prodotto durante lo Sprint, viene individuato ciò che è stato "fatto" e non, si discute dei problemi incontrati durante lo sviluppo e il **Product Owner** discute il *product backlog* fornendo una stima dei tempi di sviluppo futuri.

Sprint Retrospective Questo meeting fornisce al Team Scrum la possibilità di ispezionare se stesso e creare quindi un piano di miglioramento da attuare nelle prossime iterazioni. Durante tale meeting si analizza l'ultimo Sprint riguardo persone, processi e strumenti, così da identificare i punti di miglioramento e cosa invece ha funzionato correttamente. Lo Scrum Master fornisce un ruolo chiave durante la retrospettiva, incoraggiando il team a migliorarsi, e aiutando a trovare i punti di discussione.

2.2.3 Un metodo veloce: eXtreme Programming

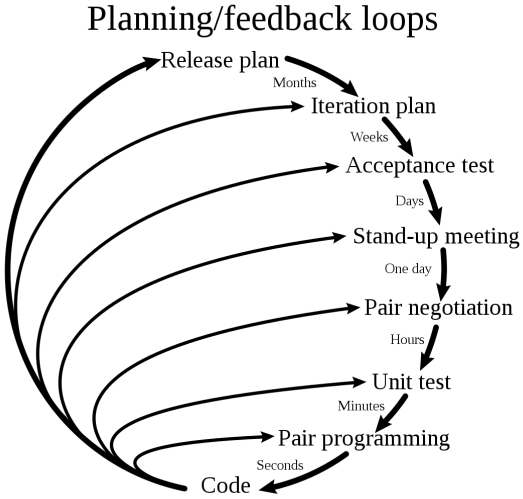

Figura 2.6: Iterazione in XP

Il metodo di sviluppo denominato **eXtreme Programming** (abbreviato **XP**), enfatizza la scrittura di codice di qualità e la rapidità di risposta ai cambiamenti dei requisiti. Rispetto a *SCRUM Framework*, XP si basa su elementi chiave totalmente differenti tra cui il *Pair Programming*, lo *Unit Testing* e il *Refactoring*, oltre che la necessità di scrivere solo codice strettamente necessario nel modo più semplice possibile.

Le 12 Regole di XP

1. **Feedback**

(a) *Pair Programming*: due programmatori lavorano assieme sulla stessa macchina, uno di essi è il *driver* che scrive il codice e l'altro è il *navigator* che ragiona sull'approccio;

(b) *Planning Game*: una riunione di pianificazione che avviene una volta per iterazione;

(c) *Test Driven Development*: i test automatici (unit e acceptance) vengono scritti *prima* del codice;

(d) *Whole Team*: il cliente non è colui che finanzia, ma colui che utilizza il sistema, motivo per cui deve essere sempre presente e disponibile per verifiche;

2. **Continuous Process**

   (a) *Continuous Integration*: integrare continuamente i cambiamenti nel codice eviterà problemi più avanti nel progetto;

   (b) *Refactoring*: riscrivere il codice senza cambiarne le funzionalità, rendendolo più semplice e generico;

   (c) *Small Releases*: il software viene rilasciato frequentemente con incrementi ridotti ma che portano valore concreto;

3. **Shared Comprehension**:

   (a) *Coding Standards*: utilizzare uno standard di scrittura preciso e condiviso, da rispettare lungo tutto il corso del progetto;

   (b) *Collective Code Ownership*: ognuno è responsabile di tutto il codice, quindi chiunque contribuisce alla sua stesura;

   (c) *Simple Design*: seguire un approccio *"simple is better"* durante la progettazione;

   (d) *System Metaphor*: descrivere formalmente il sistema mediante metafore, rendendolo più semplice da comprendere in poche parole;

4. **Programmers Wellbeing**:

   (a) *Sustainable Pace*: gli sviluppatori non dovrebbero lavorare oltre un numero di ore stabilite settimanalmente (generalmente 40).

Una delle differenze sostanziali con altri metodi *Agile*, è il focus sul **metodo di sviluppo** piuttosto che sul processo in se. Extreme Programming si prefigge quindi di ottimizzare al meglio lo sviluppo del software mediante tecniche e metodiche applicabili dagli sviluppatori stessi, senza necessità di utilizzo di figure esterne o di guide, e propone diversi concetti che sono stati ripresi in buona parte dalle metodologie *DevOps* che andremo a descrivere.

2.3 Metodologie di Sviluppo *Agile*: DevOps

In un mondo dove lo sviluppo *Agile* prende sempre più piede, si è reso necessario trovare un qualcosa che potesse migliorare notevolmente i processi già in atto, attuando delle modifiche sostanziali al come i team di sviluppo lavorano tra di loro e a quale livello di integrazione.

In particolare ciò di cui ci si è accorti nel tempo è il livello di isolamento che ogni team attua al proprio lavoro, in particolare tra i team di Development e Operations, consci però che un software è fatto di entrambe le parti che non possono prescindere dall'altra.

2.3.1 Un po' di storia

Se qualcuno chiedesse da dove ebbe **origine** il termine **DevOps**, la risposta sarebbe *Patrick Debois*. Quando fu un IT Consultant in Belgio, nel 2007, Debois lavorò ad un progetto per il governo belga riguardo una migrazione di un data center dove fu in carico del testing dei sistemi. Durante questo periodo si rese conto come stesse spendendo molto tempo tra il mondo dello sviluppo (Development) e delle Operations.

Nel 2008, durante una conferenza riguardo l'*Agile*, *Andrew Shafer* fece una sessione riguardo l'*Agile Infrastructure* dove solo *Patrick Debois* ne prese parte. Nel corso dell'anno crearono un gruppo dove poter discutere di come risolvere questo distaccamento tra il mondo Development e Operations, cosa che si evolse nell'Ottobre 2009 quando crearono un evento per avvicinare sviluppatori e system administrators che chiamarono **DevOpsDays**, unendo così le parole Development e Operations per la prima volta.

Da allora il termine **DevOps** continuò ad essere usato su Twitter, fino ad arrivare alla accezione odierna.

2.3.2 Un metodo ed un'etica

Applicare un metodo e delle pratiche *DevOps* non vuol dire solamente utilizzare determinati tool o effettuare certe azioni, ma il concetto si estende a ciò che può essere definito **una etica da applicare** a livello aziendale o progettuale a cui tutti gli attori coinvolti devono aderire per trarne il massimo beneficio.

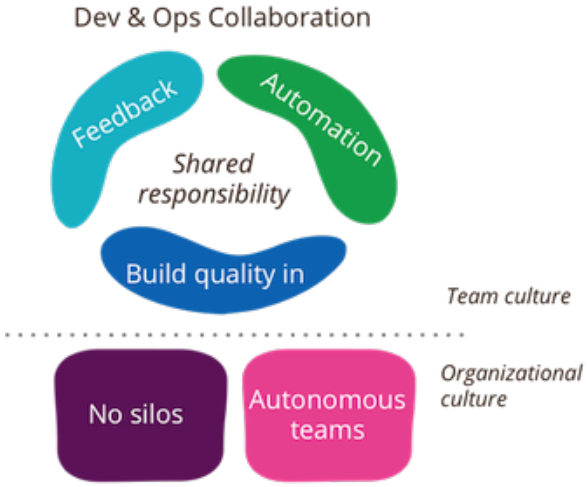

Figura 2.7: La cultura DevOps

I principi guida dell'etica *DevOps* includono un cambiamento culturale pervasivo, un metodo per misurare le prestazioni, automazione estesa e condivisione, si potrebbe anche definire il *DevOps* come l'approccio più moderno al classico **Application Lifecycle Management** (ALM). Questi principi possono essere riassunti in un framework chiamato **CALMS**, dall'acronimo *Culture - Automation - Lean Flow - Measurement - Sharing*, termine coniato da *Jez Humble*, co-autore del libro *The DevOps Handbook*.

I team di sviluppo e di operations si uniscono e diventano **responsabili dell'intero processo produttivo**, dallo sviluppo delle feature alla messa in produzione del software fino alla manutenzione e al monitoring dello stesso. L'approccio all'intero ciclo di vita del software da parte

di entrambi i team permette di avvicinare gli sviluppatori a ciò che realmente vuole l'utente, acquisendo anche feedback dalle fasi finali e aumentando la trasparenza attraverso tutti gli attori coinvolti.

2.3.3 Obiettivi delle pratiche DevOps

La principale caratteristica delle pratiche *DevOps* è la **collaborazione** tra i ruoli di Development e Operations. Per supportare tale collaborazione si necessita una attitudine di *shared responsibility*, è facile infatti perdere interesse per una materia o l'altra se ci sono team separati a gestirle senza comunicazione ne necessità di interazione tra loro, faccenda diversa quando invece il team di sviluppo controlla tutto il processo dall'inizio alla fine comprendendo processi di operations come **deployment** e **monitoring** in produzione.

Per applicare tali pratiche è inoltre indispensabile un cambiamento organizzativo all'interno dei team, **non devono esserci separazioni tra Development e Operations** sia di locazione sia di argomenti trattati, ognuno deve essere responsabile dei successi e fallimenti di un sistema puntando sempre più ad eliminare le divergenze tra i due team citati, arrivando ad un tutt'uno.

Un altro cambiamento sostanziale, derivante dalla filosofia *Agile*, è la necessità di **team autonomi**, ovvero che non necessitano la presenza costante di figure manageriali ne di processi complessi grazie al quale si possano fare determinate decisioni. Ciò si traduce in una crescente **fiducia** nelle decisioni prese dal team, ed una gestione dei processi più snello che permetta di non rallentare i lavori per colpa dell'introduzione di azioni solo managing-oriented.

Questi cambiamenti a livello culturale, pratico e metodologico si traducono in risultati tangibili a livello di business, e non solo tecnico:

- Adozione di processi *Agile* facilitata;
- Velocità di rilascio notevolmente aumentata (deliverables);
- Velocità di risposta ai cambiamenti aumentata e facilitata, aumento del livello di feedback;
- Team *cross-skilled* e *self-improved* per costi ridotti e performance migliorate;
- Qualità del prodotto notevolmente aumentata, grazie al testing estensivo.

Nel **"Atlassian Survey on DevOps Trends"** del 2020, il **99%** degli intervistati ha espresso che l'utilizzo di pratiche DevOps ha portato un **impatto positivo** nella loro realtà, tra cui il 78% ha dovuto imparare nuove skill per utilizzare tali pratiche ma al contempo al 48% ha contribuito anche ad un aumento di salario grazie al self-improvement.

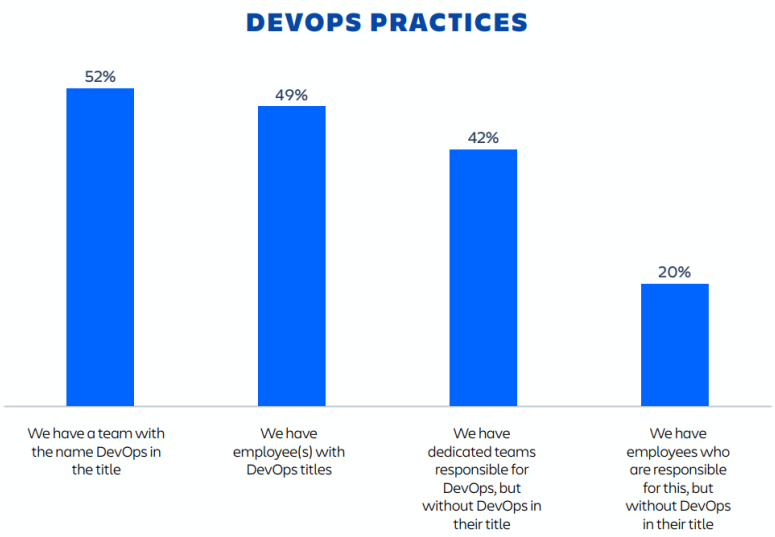

Figura 2.8: Atlassian Survey - Adozione DevOps

I vantaggi del *DevOps* sono inoltre chiari da questo survey, dove il 61% degli intervistati ha affermato che applicare tali pratiche ha migliorato la qualità dei prodotti finali, e per il 49% ha velocizzato il *time-to-market* e la consegna di *deliverables* anche parziali in meno iterazioni. Nei fattori di successo nell'implementazione di pratiche *DevOps* si trovano i **tool corretti** e le **persone giuste**, che possano collaborare e performare ottimamente con una ottima abilità di problem-solving.

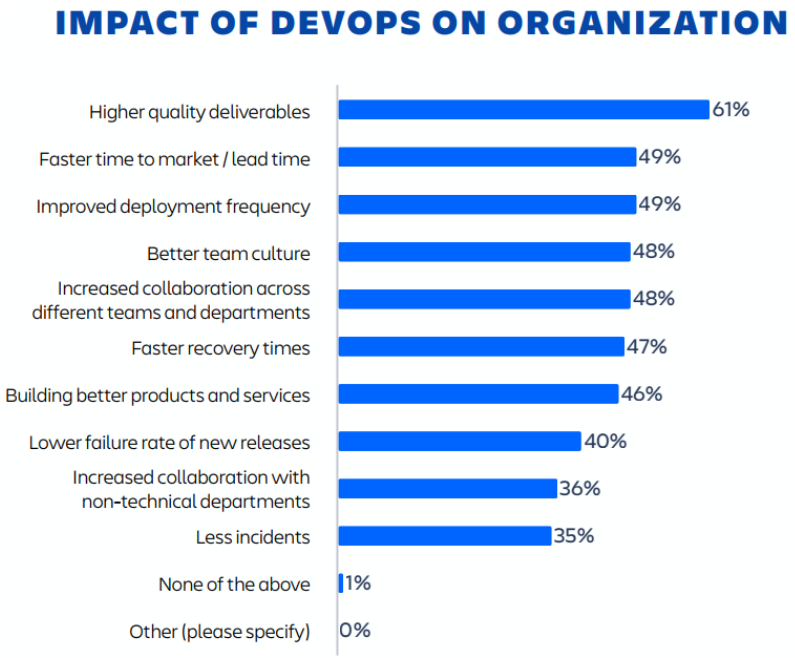

Figura 2.9: Atlassian Survey - Impatto sul Business

Il rovescio della medaglia risiede nella difficoltà di implementazione dei metodi *DevOps*, dove ben l'85% ritiene di aver trovato problemi o difficoltà, e vede come principali fattori problematici la mancanza di *skills* dei propri dipendenti, la presenza di infrastrutture *legacy* e difficilmente mantenibili ed estensibili, e una difficoltà nell'aggiustare la *cultura* aziendale

radicata nel tempo.

Infine, riprendendo i pilastri del *DevOps*, ritroviamo ciò che viene definito un "metodo" per misurare le performance e i risultati di tali pratiche, sia tecniche che business. Nel survey il 74% ritiene di avere un metodo efficace per misurare l'impatto di ciò che viene implementato (generalmente basato sulla velocità di deployment o delivery), e ben il 97% si definisce soddisfatto del modo in cui stanno misurando gli effetti delle pratiche.

2.3.4 Processo di Riferimento

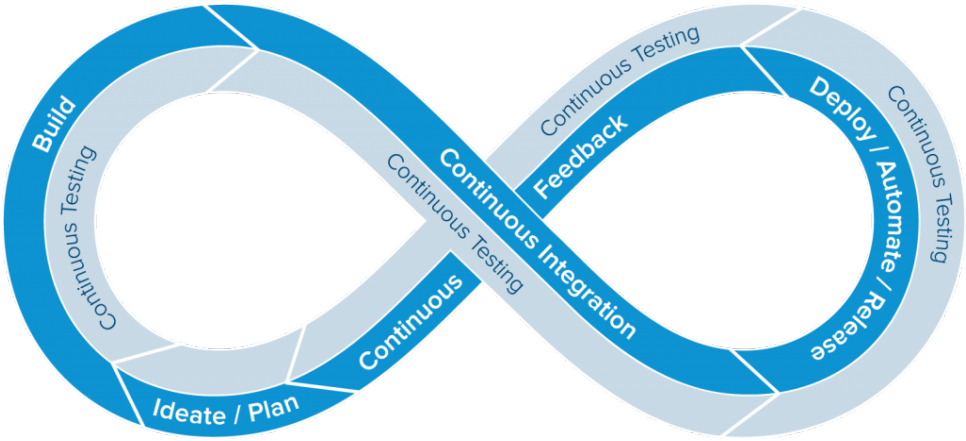

Figura 2.10: Processo DevOps-based

Una tipica modellizzazione del processo **DevOps-oriented** è definita da un simbolo dell'infinito, che riflette ciò che abbiamo introdotto con le metodologie *Agile* e lo sviluppo ad iterazioni incrementali. Il processo finale ha come scopo la massima agilità ed automazione durante tutte le fasi di sviluppo e deployment, permettendo un continuo avvicinamento tra i ruoli di development e operations in ogni fase dello stesso.

Il processo *DevOps* deve essere implementato unendo cultura, metodiche e strumenti in un ambiente coeso, basandosi sempre su processi di tipo *Agile* o sfruttando framework che permettano di implementarlo in modo efficace.

Le sue fasi sono così definite:

- **Ideate/Plan**: i tasks vengono organizzati, pianificati nell'iterazione (Sprint), e vengono inizializzati i tools di gestione del processo (Boards, Documentazione, SCM, e molti altri). L'idea di base è poter sfruttare il concetto di **user story** introdotto nelle metodologie *Agile*, così da permettere sia a "dev" che "ops" di comprendere la feature in modo semplice;

- **Code/Build**: il codice viene scritto e controllato mediante utilizzo di **Code Reviews**, che una volta approvata viene unita alla branch principale del VCS scelto. Il codice deve poi essere compilato (buildato) per controllare la sua correttezza e produrre i primi artefatti di debug;

- **Continuous Integration**: una fase cruciale che si mixa con quella di build, il codice viene testato ed integrato con l'attuale codebase prima del merge sulla branch principale, evidenziando eventuali errori nei test automatici o nelle dipendenze dei sistemi. Questo

processo è continuo e automatico per ogni commit o Code Review creati, e vengono sfruttati tool dedicati a velocizzarlo e implementarlo con successo;

- **Deploy/Release**: una volta passata la fase di *CI*, il codice è pronto per il rilascio. Questa fase viene automatizzata mediante processi di **Continuous Delivery** e **Continuous Deployment** (uno conseguente all'altro), ovvero la build degli artefatti finiti e pronti al rilascio viene "inviata" ad un sistema centralizzato di storage, per poi essere deployato automaticamente sulla infrastruttura ospitante.
- **Continuous Feedback**: la fase più operations-oriented del processo, il codice deployato deve essere controllato mediante sistemi di **Monitoring**, creando così delle metriche di performance ed implementando un flusso di **Observability**.

Nella nostra analisi ci concentreremo sul descrivere le fasi centrali del processo di *DevOps*, ovvero quelle che coinvolgono attivamente sempre tutto il team e che vengono effettuate più spesso durante le iterazioni del processo, ovvero **Continuous Integration**, **Continuous Delivery** e **Continuous Deployment**.

Continuous Integration

La *Continuous Integration (CI)* è la pratica di automatizzare ed integrare i cambiamenti del codice da parte di molte fonti (sviluppatori) in una singola codebase. Viene definita come una delle pratiche fondamentali del processo *DevOps* e permette agli sviluppatori di scrivere, integrare e testare velocemente le feature utilizzando dei sistemi di automazione che controllano la correttezza del software on-demand.

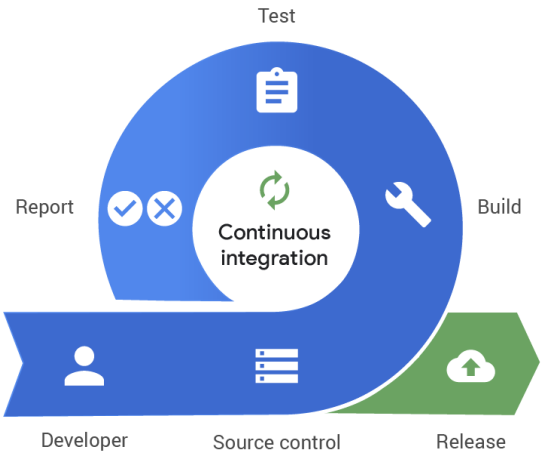

Figura 2.11: DevOps - Continuous Integration

Per implementare un sistema di *CI*, sono necessari:

1. Un sistema di repository versioning (generalmente Git);
2. Un codice auto-testante, ovvero che contiene test automatici (Unit, Integration, E2E);
3. Un ambiente di test unificato e indipendente dalle macchine degli sviluppatori;
4. Un tool che gestisca il processo (Jenkins, Gitlab CI, e molti altri);

5. Massima visibilità dei risultati prodotti dal processo (test, errori, dipendenze);

Il processo prevede diverse fasi che includono la cooperazione tra sviluppatori e un tool che gestisca il processo stesso, implementando le sue fasi automatiche:

1. **Developer**: lo sviluppatore scrive il codice della feature necessaria, assieme al codice è necessario scrivere **test automatici** sia di tipo *unit* che *integration*. La presenza di questi test valuterà la qualità finale del processo di integrazione;

2. **Source Control**: le modifiche alla code base vengono inserire in un sistema di version control (es. Git), generalmente usando *branch* separate dalla *master*;

3. **Build**: il tool di CI avvia una build automatica comprendente le modifiche aggiunte dallo sviluppatore al codice principale. L'ambiente di build è controllato ed unificato, così da ridurre al minimo la variabilità;

4. **Test**: il tool di CI avvia il *testing automatico* sulla codebase, che comprende sia *unit tests* che *integration tests*;

5. **Report**: il risultato dei test avviati viene visualizzato ed analizzato all'interno del tool di CI, le parti coinvolte vengono informate dello status dell'integrazione e di eventuali fallimenti.

Questo processo viene iterato fino alla fine dello sviluppo della feature, ogni singola modifica sarà quindi controllata dal sistema e solo se tutto passerà i test automatici allora si potrà integrare con successo il codice nuovo nella codebase principale (generalmente la *master* branch). Una delle pratiche comuni durante l'utilizzo di una pipeline di CI, è l'introduzione del **TDD** come visto nei principi di *XP*, ottenendo così una copertura completa delle righe di codice rispetto ai test scritti.

Una estensione naturale del processo di *CI* si presenta con l'introduzione dell'**analisi statica e dinamica** del codice mediante lo stesso tool o altri tools dedicati. Questa analisi permette di ridurre l'impatto del *technical debt* in fasi precoci dello sviluppo, fornendo allo sviluppatore uno sguardo veloce in cosa potrebbe potenzialmente causare bug o problemi di mantenibilità nel lungo termine e risolverli in fasi precoci dello sviluppo.

Continuous Delivery

Se con la *Continuous Integration* abbiamo descritto un processo in continuo utilizzo attivo da parte di tutti gli sviluppatori, il processo di *Continuous Delivery* può essere approcciato con diverse metodologie e tempistiche in base al proprio ciclo di rilascio del software.

Nella fase finale dello schema di CI, troviamo la fase di **Release**, generalmente preceduta da ciò che viene definita la **delivery degli artefatti software**, ovvero la build automatica e l'invio con storage dei prodotti finali pronti all'uso e alla release in una fase successiva. Un tipico artefatto in output di un sistema di CD è un *container*, unità che permette poi di semplificare anche il deployment successivo in ambienti diversi.

L'obiettivo di questa processo è di avere una collezione di artefatti software **pronti alla release** in qualsiasi momento, così da velocizzare il processo di deployment successivo. L'evento che può triggerare una build automatica potrebbe essere un *tag* sul *repository*, oppure un trigger

Figura 2.12: DevOps - Processo Completo

manuale attraverso un tool dedicato, oppure la semplice fine di una iterazione di sviluppo (Sprint) dopo la quale si necessita di inviare il software prodotto al cliente o di "congelare" la codebase attuale.

Continuous Deployment

Un flusso di CI/CD maturo sfocia in quello che viene definito *Continuous Deployment*, ovvero l'automazione dell'ultimo tratto che porta alla fruizione automatica delle nuove feature sviluppate, direttamente in un ambiente di produzione finale.

Questo processo fa forte affidamento nella solidità e precisione dei precedenti flussi, dove il testing deve necessariamente essere estensivo al punto di garantire un rilascio in produzione senza problemi, che possa così essere automatizzato senza preoccupazioni. Generalmente l'uso del deployment automatico porta al *time-to-market* più veloce possibile, e una feature potrebbe in pochi minuti essere operativa dopo l'esser stata sviluppata e testata.

Una feature indispensabile a supporto del *Continuous Deployment* è la possibilità di effettuare dei **rollback** senza impattare il resto del sistema ed in modo automatico in caso di fallimenti non previsti. Questi avvisi di "non funzionamento" devono necessariamente provenire da un sistema di *monitoring* che permetta di valutare errori e prestazioni del software in una fase post-release.

2.4 *DevOps*: l'impatto sul business

I dati pubblicati dal *State of DevOps Report* redatto da **Puppet** e **CircleCI**, due dei più grandi strumenti utilizzati in ambito DevOps, rivelano gli impatti significativi che tali pratiche hanno portato in ambienti business ed IT-oriented.

Dopo aver intervistato oltre 2500 partecipanti, nonostante le condizioni lavorative dell'anno 2020 dovute alla pandemia globale di COVID-19, sono emersi alcuni punti chiave nel mondo DevOps:

DevOps come Piattaforma Con la crescente necessità di unificare i team, si sono creati dei team "di prodotto" che si occupano di tutto il ciclo di vita di un software. La nascita di molti team separati rischia di creare diversità e problematiche per l'uso di tools, pratiche e processi diversi.

Molte realtà si sono quindi adattate nel creare un **team di platform** che possa gestire tutti questi aspetti per tutti i team in modo univoco, unificando i processi e strumenti, e snellendo il

peso dell'uso degli stessi da parte di ogni sviluppatore, che prende parte attivamente al processo ma non si trova in mezzo a decisioni burocratiche che potrebbero rallentare il lavoro.

Oltre il 63% delle aziende ha una piattaforma interna comune, e il 77% ha 3 o più servizi all'interno della stessa. Da denotare inoltre il grado di utilizzo di tale piattaforma, tendente sempre più verso il massimo, ma ancora più del 50% non la sfrutta a dovere (< 50% degli sviluppatori in azienda).

Trattare la piattaforma DevOps come un **prodotto** è una delle core features di una azienda di successo, quindi mantenere dei requisiti per gli stakeholders con una roadmap, gestire l'onboarding nella stessa in modo efficace ed avere una figura di riferimento come un Product Manager che possa analizzare i requisiti e il "mercato".

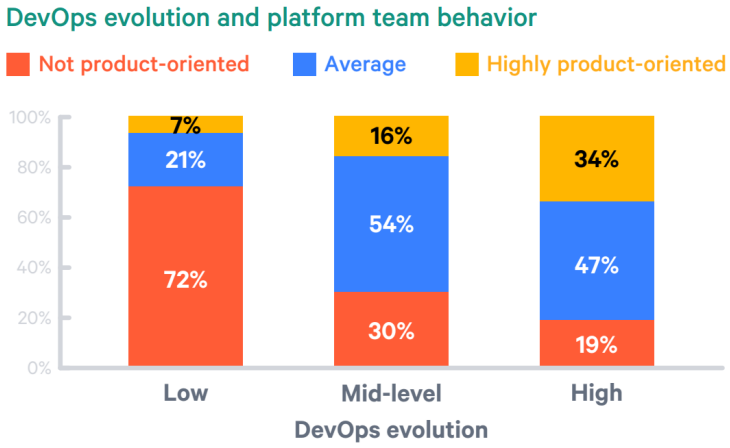

Figura 2.13: State of DevOps 2020 - Platform Behaviour

Cambiamenti nel Management Se una azienda non è in grado ancora di trattare il DevOps come una piattaforma unica, il metodo più efficace per velocizzare la delivery del software è un cambiamento di **cultura ed etica** a livello di management.

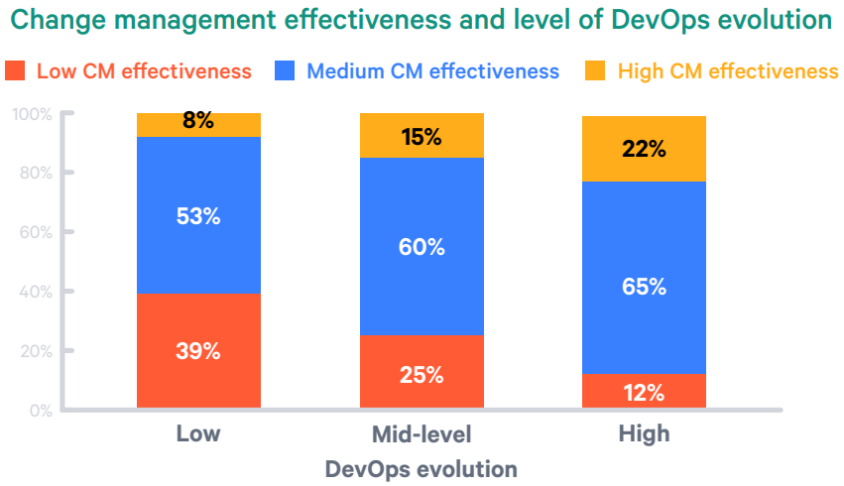

Figura 2.14: State of DevOps 2020 - Change Management

Si sono visti 4 approcci principali al cambiamento interno:

- *Ad-Hoc*: automazione ridotta, risk-mitigation ridotta, improntata allo small/mid market e adattabilità ridotta nel tempo;
- *Governance Focused*: automazione ridotta, risk-mitigation ridotta, improntata al mid/large market e dipendente da politiche aziendali interne;
- *Engineering Driven*: alto livello di automazione, risk-mitigation estesa, improntata allo small/mid market con alta adattabilità nel tempo;
- *Operationally Mature*: alto livello di automazione, risk-mitigation estesa, improntata al mid market specificatamente e dipendente dalle politiche aziendali interne.

Si è anche notato come l'approccio più governance-oriented e quindi più rispondente alle politiche aziendali preesistenti, rallenti notevolmente i processi e renda inefficiente il lavoro allontanandosi così dal concetto di *Agile Development*. Un alto livello di automazione al contrario, rende gli addetti ai lavori molto confidenti nel cambiamento in atto, riduce i rischi dovuti a problematiche di ogni natura e assicura un *time-to-market* nettamente migliore.

Ogni approccio presenta comunque un lato negativo, ad esempio le aziende *Operationally Mature* presentano una tolleranza al rischio molto bassa, mentre le *Engineering Driven* si sentono limitate dalla loro architettura applicativa. Non esiste l'approccio perfetto in nessuna delle 4 casistiche, ma i driver di successo sono sicuramente un processo più snello, meno governance e un livello di automazione tale per cui la confidenza possa rimanere alta sia nello sviluppo che nel management.

> *The problem isn't change, per se, because change is going to happen; the problem, rather, is the inability to cope with change when it comes.*
>
> Kent Beck - Extreme Programming Explained: Embrace Change,

2.4.1 Il *ROI* del DevOps

Una tipica domanda che qualsiasi leader aziendale si pone quando si approccia alle metodologie *DevOps* è: quanto ritorno positivo (economico e tecnico) mi comporta, ed in quanto tempo? Risulta chiaro dalla teoria come l'uso di queste pratiche comporti uno sviluppo più *Agile*, rischi ridotti nel medio-lungo termine e *time-to-market* più veloce, ciò che manca è una metrica per valutare quantitativamente l'impatto sull'azienda e sui prodotti offerti dalla stessa.

Il calcolo del ROI si basa principalmente sui guadagni attesi rispetto alle spese, solitamente espressi in guadagni economici ma spesso scartando il guadagno in fattore *tempo*, uno dei punti di forza di queste pratiche. L'analista **Michael Cote** ha affermato come definire quantitativamente il ROI del *DevOps* sia contemporaneamente "assurdo e fondamentale" per la sua complessità in fatto di variabili in gioco, che renderebbe la stima estremamente imprecisa e mai veritiera, quindi difficilmente stimabile.

Esiste però un indicatore numerico che può riflettere il principale vantaggio del *DevOps*, ovvero il cosiddetto **MTTR** - *mean time to repair* - il tempo impiegato a rilevare e risolvere gli errori del software o del progetto in se, proporzionale al tempo di inattività o stallo del ciclo di sviluppo, valore che diminuisce con meno *deployments* effettuati.

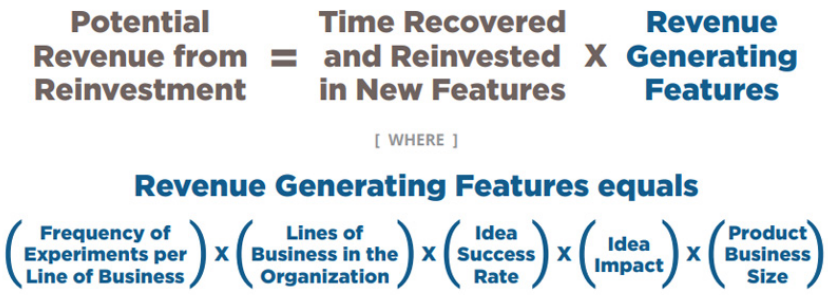

Figura 2.15: DevOps - Impatto sul ROI

Cosa si può dire riguardo l'impatto nel lungo e medio termine? CloudMunch, per esempio, misura annualmente valori come il tempo di deployment, il livello di automazione di task ripetitivi e del supporto, e moltiplicandoli per il costo medio di uno sviluppatore all'ora ricava una misura potenziale del ROI del *DevOps* nel corso di un anno solare. Esistono diversi esempi di metodologie diverse, tutte però adattate alla propria realtà e mai un indicatore preciso ed univoco del vero ROI.

Non vi è ombra di dubbio sulla difficoltà di calcolo del ROI, specialmente per un processo che tutt'ora è in fase di adozione e non è improntato tanto sull'aumento del guadagno economico ma più sul velocizzare quello previsto dal progetto. Secondo *Patrick Debois*, uno dei padri fondatori del movimento *DevOps*, dovremmo pensare il ROI come una "realizzazione più veloce dei benefici" già previsti dal progetto in se, accorciando così i cicli di sviluppo che possono essere ridotti e raggiungere gli obiettivi prima.

Capitolo 3

Sviluppo ed Automazione su Cloud

3.1 Perchè il Cloud?

3.2 *Infrastructure-as-a-Code*

3.3 *Configuration-as-a-Code*

3.4 Containers ed ambienti controllati

Capitolo 4

Analisi del Processo di Sviluppo

4.1 Struttura del Progetto

4.2 Il processo di sviluppo

4.3 Requisiti e KPI

4.4 Il processo DevOps

4.4.1 Architettura High-Level e Fasi

4.4.2 Gestione del Codice

4.4.3 Pull Requests e Code Review

4.4.4 Continuous Integration

Test Automation

Code Analysis

4.4.5 Continuous Delivery

Build Automation

Artifacts Delivery

4.4.6 Deployment

4.5 Tecnologie e Strumenti

4.5.1 SCM: Git

4.5.2 Build System: Bazel

4.5.3 Cloud Provider: AWS

4.5.4 PR Management: Phabricator ed Arcanist

4.5.5 CI/CD: Jenkins

4.5.6 Code Analysis: SonarQube

Capitolo 5

Tecnologie di Background

5.1 Cloud Provider: *AWS*

5.2 Infrastructure-as-a-Code: *Terraform*

5.3 Configuration-as-a-Code: *Ansible*

5.4 Container Engine: *Docker*

Capitolo 6

Architettura Cloud

6.1 Diagramma Architetturale

6.2 Configurazione di Phabricator

6.3 Configurazione di Jenkins

6.3.1 Agent su AWS EC2

6.3.2 Agent macOS On-Premise

6.4 Configurazione di SonarQube

6.5 Creazione ambiente di build con Docker

Capitolo 7

La Pipeline di CI

7.1 Tecnologie e Strumenti

7.1.1 Phabricator ed Arcanist: un flusso controllato

7.1.2 Jenkins: il motore del processo

7.1.3 Docker: ambiente di testing unificato

7.1.4 SonarQube: controllo qualità

7.2 Le fasi della Pipeline

7.3 Analisi del Codice

7.3.1 Quality Gates

7.3.2 Code Coverage

7.3.3 Risposta a cambiamenti nella qualità

7.4 Risultati di Testing e QA

Capitolo 8

La Pipeline di CD

8.1 Tecnologie e Strumenti

8.1.1 Jenkins: il motore del processo

8.1.2 Docker: ambiente di build unificato

8.2 Le fasi della Pipeline

8.3 Analisi delle Vulnerabilità

8.4 Risultati

Capitolo 9

Conclusioni

9.1 Obiettivi Raggiunti

9.2 Risultati su Requisiti e KPI

9.3 Evoluzioni Future

9.4 Considerazioni Personali