

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

Relatore: Prof. Mariani Leonardo

Co-relatore: Dott. Mesiano Cristian

Relazione della prova finale di:
Renzo Simone
Matricola 781616

Anno Accademico 2019-2020

*Ai miei genitori, scusandomi per l'enorme ritardo.
Alla mia fidanzata, per essermi stata sempre a fianco in un anno veramente difficile.
Ai miei amici, per aver gufato sempre e comunque.
Ai miei colleghi grazie al quale non sarebbe stata possibile questa opportunità.
A me stesso, per non aver smesso di crederci.*

*Un ringraziamento particolare al Dott. Cristian Mesiano per aver creduto
ed investito in me fin dall'inizio.*

Abstract

In un mondo in continuo sviluppo, la necessità di adattarsi velocemente al cambiamento è spesso ciò che permette di contraddistinguere le realtà di successo dalle fallimentari. Il software è probabilmente uno dei prodotti che più segue questa filosofia di cambiamento repentino, con il rilascio continuo 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.* ”

Leon C. Megginson, 1963, citando C. Darwin

Indice

Introduzione	7
Il Contesto Aziendale	7
Scopo del Project Work	8
1 Modelli di Sviluppo del Software	10
1.1 Il ciclo di vita del software	10
1.1.1 Le fasi del ciclo di vita	10
1.1.2 I modelli classici: Waterfall e Iterativo	11
1.2 Modelli <i>Agile</i>	13
1.2.1 La filosofia <i>Agile</i>	14
1.2.2 Un caso di successo: SCRUM Framework	15
1.2.3 Un metodo veloce: eXtreme Programming	17
1.3 Metodologie di Sviluppo <i>Agile</i> : DevOps	18
1.3.1 Un po' di storia	19
1.3.2 Un metodo ed un'etica	19
1.3.3 Obiettivi delle pratiche DevOps	20
1.3.4 Processo di Riferimento	22
1.4 <i>DevOps</i> : l'impatto sul business	25
1.4.1 Il <i>ROI</i> del DevOps	27
2 Sviluppo ed Automazione su Cloud	29
2.1 Introduzione	29
2.2 <i>Infrastructure-as-Code</i>	31
2.3 <i>Configuration-as-Code</i>	32
2.4 <i>Containers</i> ed ambienti controllati	33
3 Analisi del Processo di Sviluppo	35
3.1 Struttura del Progetto	35
3.2 Il processo di sviluppo	36
3.3 Requisiti e KPI	37
3.4 Analisi del Processo Legacy	38
3.5 Il processo DevOps	39
3.5.1 Architettura High-Level e Fasi	39
3.5.2 Gestione del Codice	39
3.5.3 Pull Requests e Code Review	40
3.5.4 Continuous Integration	41
3.5.5 Continuous Delivery	41
3.5.6 Deployment	42

3.6	Tecnologie e Strumenti	42
3.6.1	SCM: Git	42
3.6.2	Build System: Bazel	42
3.6.3	Cloud Provider: AWS	42
3.6.4	Orchestrator: Phabricator ed Arcanist	43
3.6.5	CI/CD: Jenkins	43
3.6.6	Code Analysis: SonarQube	43
3.7	Analisi del Processo <i>DevOps</i>	45
4	Tecnologie di Background	46
4.1	Cloud Provider: <i>AWS</i>	46
4.2	Infrastructure-as-Code: <i>Terraform</i>	47
4.2.1	Esempio: Creazione di una VM	48
4.3	Configuration-as-Code: <i>Ansible</i>	49
4.3.1	Esempio: installazione di Apache Httpd	50
4.4	Container Engine: <i>Docker</i>	51
4.4.1	Esempio: applicazione Java	52
5	Architettura Cloud	53
5.1	Introduzione e Definizione	53
5.2	Diagramma Architetturale	54
5.3	Creazione Infrastruttura	57
5.3.1	Struttura <i>IaC</i>	57
5.3.2	Struttura <i>CaC</i>	59
5.4	Configurazione di Bazel Cache	61
5.5	Configurazione di Phabricator	62
5.6	Configurazione di Jenkins	63
5.6.1	Agent su AWS EC2	64
5.6.2	Agent macOS On-Premise	65
5.7	Configurazione di SonarQube	65
5.8	Creazione ambiente di build con Docker	67
6	La Pipeline di CI	68
6.1	Tecnologie e Strumenti	68
6.1.1	Phabricator ed Arcanist: un flusso controllato	68
6.1.2	Jenkins: il motore del processo	69
6.1.3	Docker: ambiente di testing unificato	71
6.1.4	SonarQube: controllo qualità	71
6.2	Diagramma della Pipeline	73
6.2.1	Flusso di Sviluppo	73
6.2.2	<i>Continuous Build</i>	74
6.2.3	Analisi del Codice	75
6.3	Testing e Analisi del Codice	75
6.3.1	Unit e Integration Tests	75
6.3.2	Code Coverage	76
6.3.3	Quality Gates	76
6.3.4	Risposta a cambiamenti nella qualità	77
6.4	Risultati	78
6.5	Esperienza Applicativa	78

7 La Pipeline di CD	80
7.1 Tecnologie e Strumenti	80
7.1.1 Git e Bitbucket: il <i>trigger</i>	80
7.1.2 Jenkins: il motore del processo	80
7.1.3 AWS CodeBuild: compilazioni remote	82
7.1.4 Docker: ambiente e artefatti	82
7.2 Diagramma della Pipeline	83
7.3 Analisi delle Vulnerabilità	83
7.4 Risultati	84
7.5 Esperienza Applicativa	84
Conclusioni	85
Bibliografia	87

Elenco delle figure

1	Sphere - Architettura High Level	7
2	Project Work Gantt	9
1.1	Software Development Lifecycle - Source: Arkbauer	10
1.2	Modello Waterfall - Source: Justin Jones, Scott Wardell	11
1.3	Modello Iterativo - Source: SlideTeam	12
1.4	Modello Agile Generico - Source: Capterra	13
1.5	SCRUM Framework - Source: Scrum.org	15
1.6	Iterazione in XP - Source: Wikipedia	17
1.7	La cultura DevOps - Source: Martin Fowler	19
1.8	Atlassian Survey - Adozione DevOps	21
1.9	Atlassian Survey - Impatto sul Business	21
1.10	Processo DevOps-based - Source: HiClipart	22
1.11	DevOps - Continuous Integration - Source: PagerDuty	23
1.12	DevOps - Processo Completo - Source: RedHat	25
1.13	State of DevOps 2020 - Platform Behaviour	26
1.14	State of DevOps 2020 - Change Management	26
1.15	DevOps - Impatto sul ROI - Source: Veritis	28
2.1	La Piramide del Cloud - Source: Vladimir Fedak	30
2.2	Infrastructure-as-Code - Source: HashRoot	31
2.3	Dalle VM ai Container - Source: Docker	33
2.4	Contenuto di un Container - Source: AWS	34
3.1	Sphere - Architettura	35
3.2	Sphere - Workflow Legacy	36
3.3	Sphere - Workflow DevOps	40
3.4	Phabricator Web UI	43
3.5	Jenkins Web UI	44
3.6	SonarQube Web UI	44
4.1	Logo Amazon Web Services - Source: AWS	46
4.2	Servizi AWS (principali) - Source: AWS	46
4.3	Terraform - Schema Funzionamento - Source: HashiCorp	47
4.4	Ansible - Schema Funzionamento - Source: Edureka!	50
4.5	Docker - Schema Funzionamento - Source: Docker	51
5.1	DevOps - Architettura Cloud - Esterna	54
5.2	DevOps - Architettura Cloud - Interna	55
6.1	Sphere CI - Development Flow	73

6.2	Sphere CI - Continuous Build	74
6.3	Sphere CI - Code Analysis	75
6.4	Jenkins - Tests Analyzer Plugin	76
6.5	SonarQube - Quality Gate	77
6.6	SonarQube - Issues	77
7.1	Sphere - Continuous Delivery	83

Elenco delle tabelle

3.1 Requisiti e KPIs - Workflow Legacy	38
3.2 Requisiti e KPIs - Workflow DevOps	45
6.1 Sphere - Raggiungimento Obiettivi CI	78
7.1 Sphere - Raggiungimento Obiettivi CD	84

Introduzione

Il Contesto Aziendale

Storia

PerceptoLab S.r.l. è una startup italiana (con seconda filiale in Germania) con sede a Giussano (MB), fondata nel 2017. L'azienda si è da sempre occupata di innovazione in ambito *insurtech*, grazie a tecnologie proprietarie nell'ambito della *Computer Vision* e dell'*Intelligenza Artificiale*. Dal 2019 PerceptoLab si è concentrata sullo sviluppo dell'ecosistema *Sphere*, una soluzione all-round per la gestione del *claim* assicurativo e la digitalizzazione della propria abitazione. Il progetto consta inoltre di 2 brevetti depositati (*Sphere Vision* e *Sphere Index*).

L'azienda è ad oggi composta da 11 dipendenti, ad esclusione del *CEO Dott. Mesiano Cristian*, divisi in 4 team diversi in base alla branca di lavoro: Mobile, Backend, Computer Vision e Deep Learning.

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.

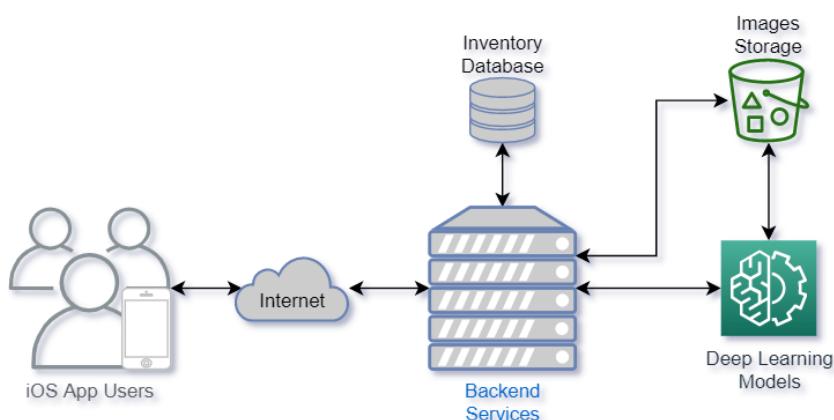


Figura 1: Sphere - Architettura High Level

Grazie a *Sphere*[1] 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 personalizzare la propria esperienza integrando i moduli necessari al proprio business, senza preoccuparsi di dipendenze esterne o di costi non controllabili.

L'architettura di Sphere, descritta ad alto livello in figura 1, segue la filosofia client-server, con una *App iOS* che funge da "frontend" ed un backend a *microservizi*. Il backend interagisce con il database del Digital Inventory, gestendo le entità individuate dal sistema o dall'utente, e con l'object storage contenente le immagini da dove i *modelli di AI* traggono le loro analisi.

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.

Scopo del Project Work

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 i seguenti 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.

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).

October				November				December			
W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4
Analisi Requisiti											
	Infrastruttura Cloud										
	Implementazione Pipeline CI				Implementazione Pipeline CD				Implementazione Code Analysis		
									Training Interno Utilizzo		

Figura 2: Project Work Gantt

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 1

Modelli di Sviluppo del Software

1.1 Il ciclo di vita del software

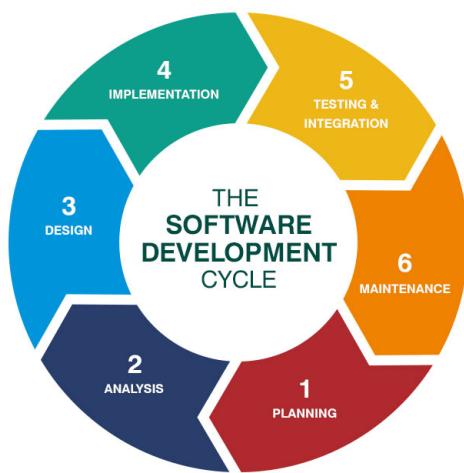


Figura 1.1: Software Development Lifecycle - **Source:** Arkbauer

1.1.1 Le fasi del ciclo di vita

Il ciclo di vita del software[2], come descritto in figura 1.1, 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 sviluppo software. 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.

1.1.2 I modelli classici: Waterfall e Iterativo

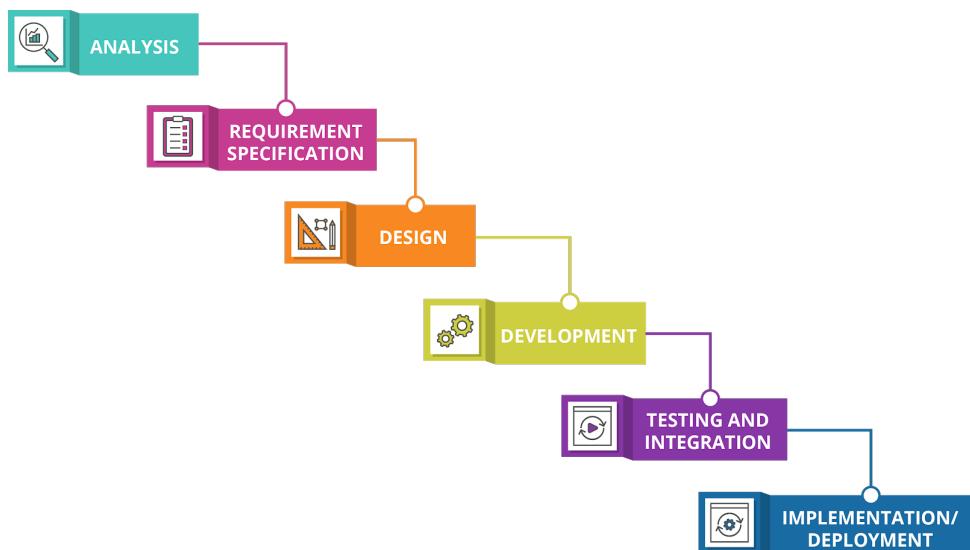


Figura 1.2: Modello Waterfall - **Source:** Justin Jones, Scott Wardell

Waterfall Il processo di sviluppo storicamente più tradizionale e semplice è chiamato *Waterfall*[3]. 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 manifatturiera 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;

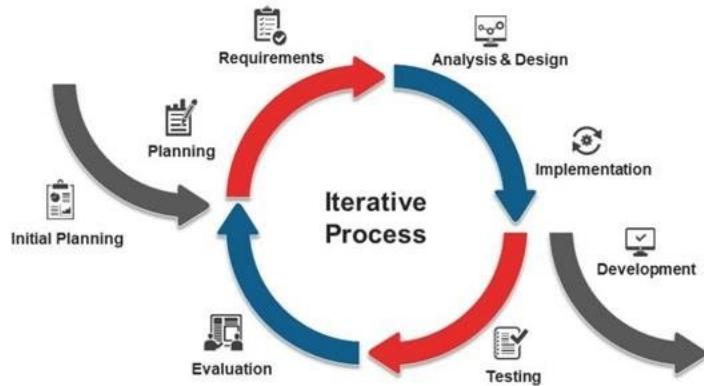


Figura 1.3: Modello Iterativo - **Source:** SlideTeam

Iterativo Una evoluzione del processo *Waterfall* è il modello *Iterativo*[4], 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.

1.2 Modelli Agile

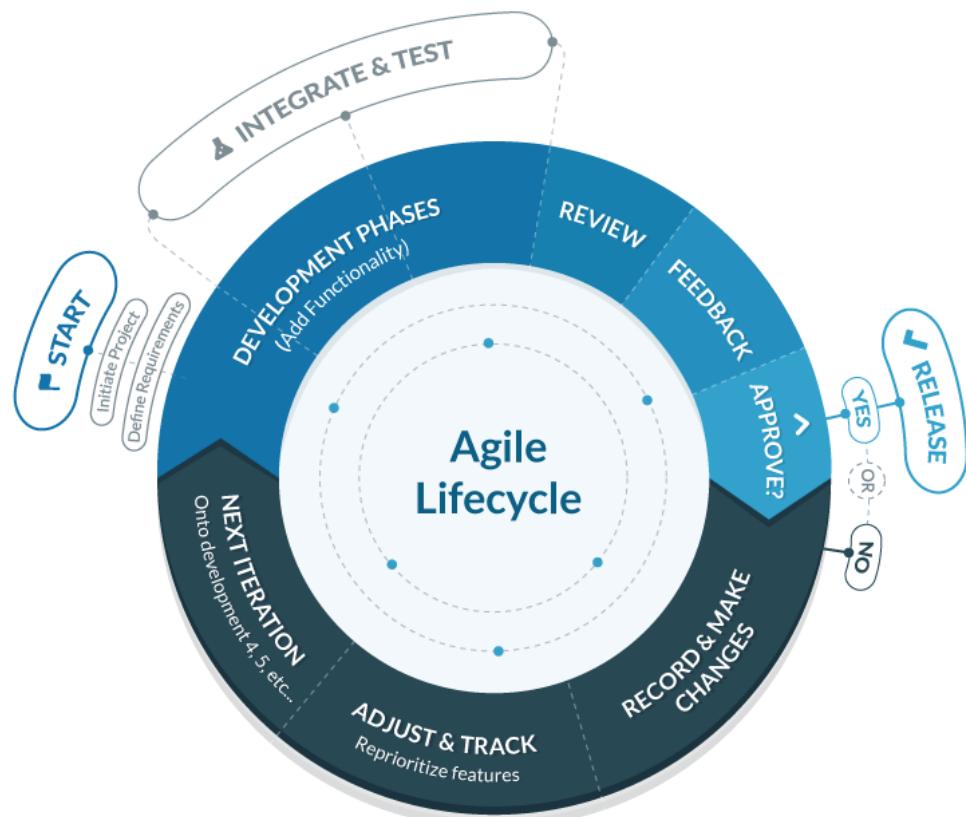


Figura 1.4: Modello Agile Generico - Source: Capterra

1.2.1 La filosofia *Agile*

La filosofia *Agile* è emersa a partire dai primi anni 2000, grazie alla pubblicazione del "Manifesto for Agile Software Development"^[5] nel 2001, nato dall'evoluzione dei metodi classici iterativi ed incrementalni.

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.

1.2.2 Un caso di successo: SCRUM Framework

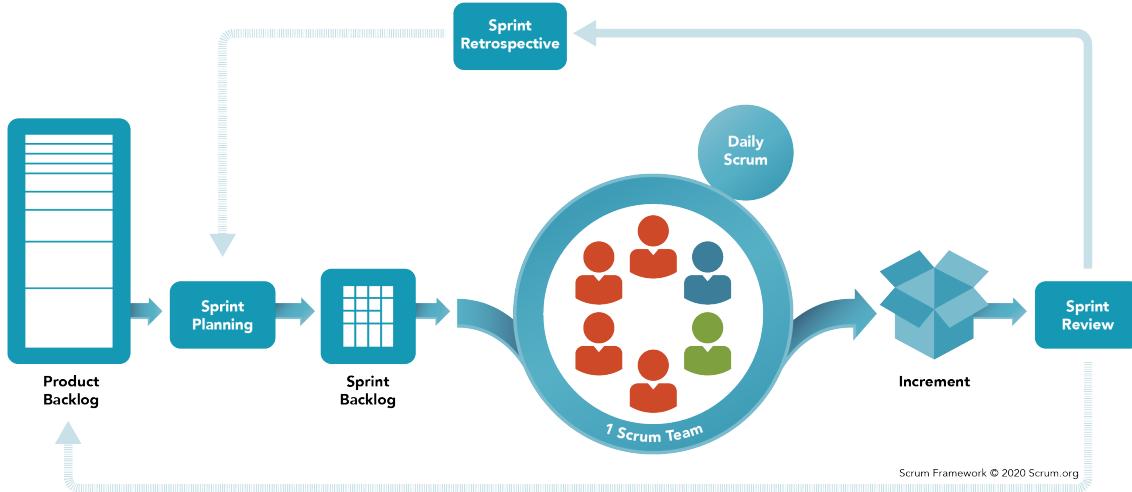


Figura 1.5: SCRUM Framework - Source: Scrum.org

Nel mondo *Agile* sono nati diversi metodi e processi con approcci differenti alla filosofia precedentemente descritta, uno dei quali è **SCRUM Framework**[6], 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 *springano* 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 ispira alla 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 devono essere effettuate da ispettori qualificati;
3. **Adattamento:** se durante l'ispezione si rilevano 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 tenuti 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 né 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.

1.2.3 Un metodo veloce: eXtreme Programming

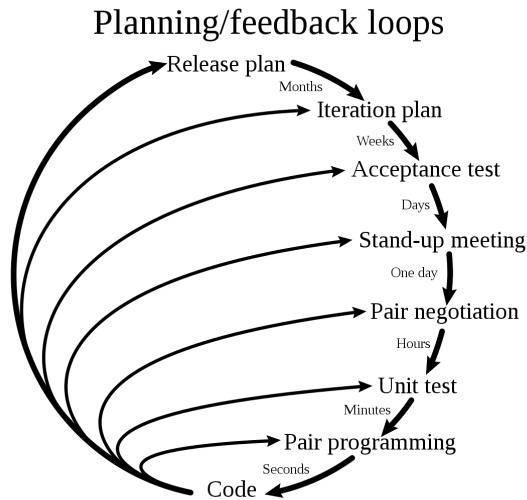


Figura 1.6: Iterazione in XP - Source: Wikipedia

Il metodo di sviluppo denominato **eXtreme Programming**[7] (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 programmati 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.

1.3 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.

1.3.1 Un po' di storia

Se qualcuno chiedesse da dove ebbe **origine**[8] il termine **DevOps**, la risposta sarebbe *Patrick Debois*. Quando lavorò come 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* a cui solo *Patrick Debois* partecipò. 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.

1.3.2 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 definita **un'etica da applicare** a livello aziendale o progettuale a cui tutti gli attori coinvolti devono aderire per trarne il massimo beneficio.

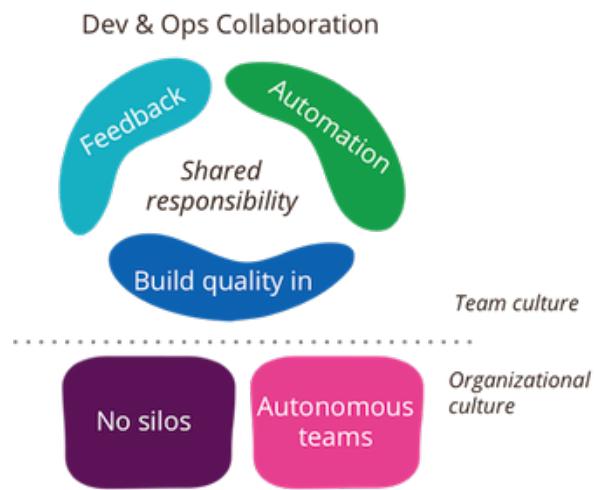


Figura 1.7: La cultura DevOps - **Source:** Martin Fowler

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**[9], 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.

1.3.3 Obiettivi delle pratiche DevOps

La principale caratteristica delle pratiche *DevOps*[10] è la **collaborazione** tra i ruoli di Development e Operations. Per supportare tale collaborazione viene necessaria 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 né 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 né 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"[11] 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 per il **48%** ha contribuito anche ad un aumento di salario grazie al self-improvement.

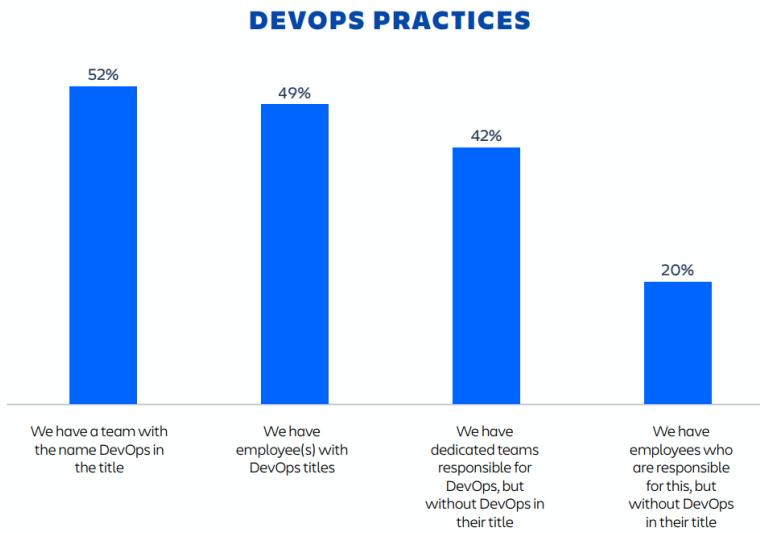


Figura 1.8: Atlassian Survey - Adozione DevOps

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.

IMPACT OF DEVOPS ON ORGANIZATION

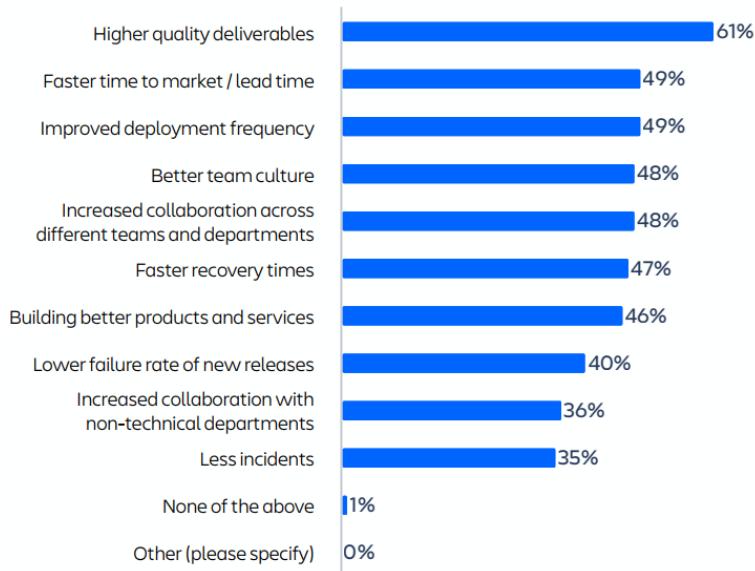


Figura 1.9: Atlassian Survey - Impatto sul Business

Il rovescio della medaglia risiede nella difficoltà di implementazione dei metodi *DevOps*, dove ben l'85% ritiene di aver riscontrato 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 difficoltà nel trasformare una *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 di tali pratiche.

1.3.4 Processo di Riferimento

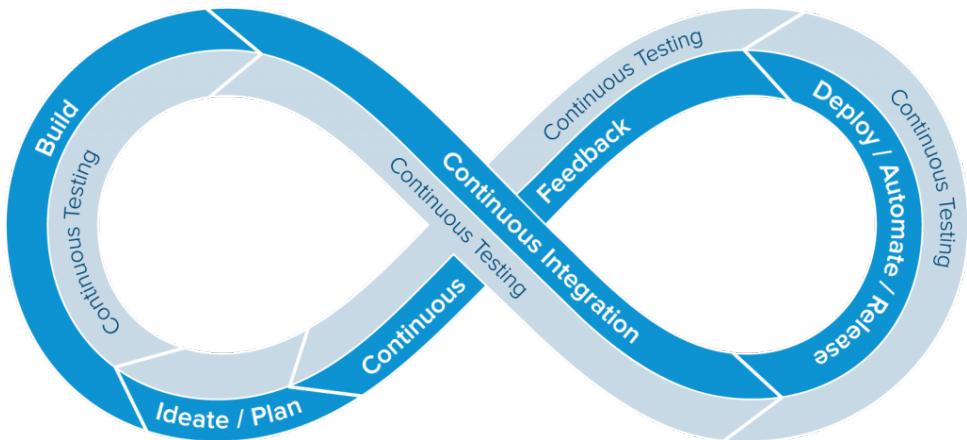


Figura 1.10: Processo DevOps-based - **Source:** HiClipart

Una tipica modellizzazione del **processo DevOps-oriented**[12] è definita da un simbolo dell'infinito (figura 1.10), 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 fonde 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 *DevOps*, 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)*[13] 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.

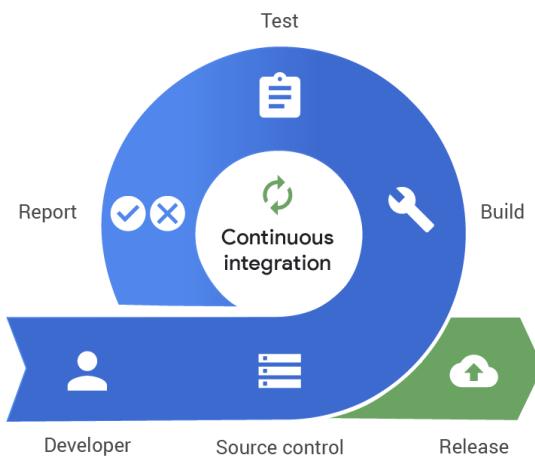


Figura 1.11: DevOps - Continuous Integration - **Source:** PagerDuty

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 inserite 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 stato 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, risolvendoli 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*[14] può essere approcciato con diverse metodologie e tempistiche in base al proprio ciclo di rilascio del software.

Nella fase finale dello schema di CI (figura 1.11), troviamo la fase di **Release**, generalmente preceduta da ciò che viene definita la **delivery degli artefatti software**, ovvero la build automatica e lo 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 questo 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 scatenare una build automatica potrebbe essere un *tag* sul *repository*, oppure un avvio



Figura 1.12: DevOps - Processo Completo - **Source:** RedHat

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*[15], ovvero l'automazione dell'ultimo tratto che porta alla fruizione automatica delle nuove feature sviluppate, direttamente in un ambiente di sviluppo, staging o 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.

1.4 DevOps: l'impatto sul business

I dati pubblicati dal *State of DevOps Report*[16] 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".

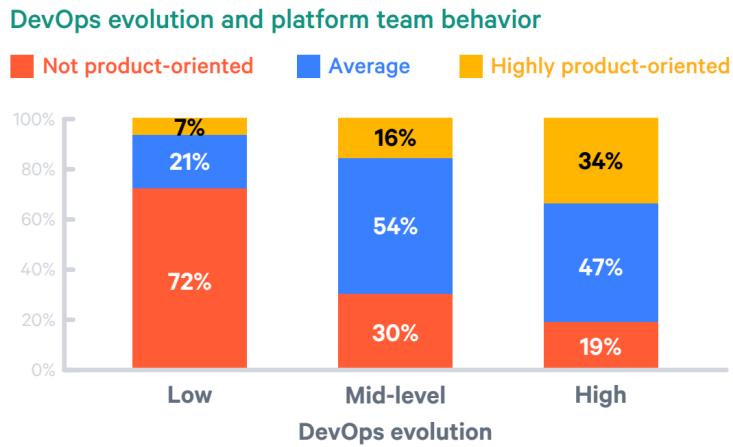


Figura 1.13: State of DevOps 2020 - Platform Behaviour

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.

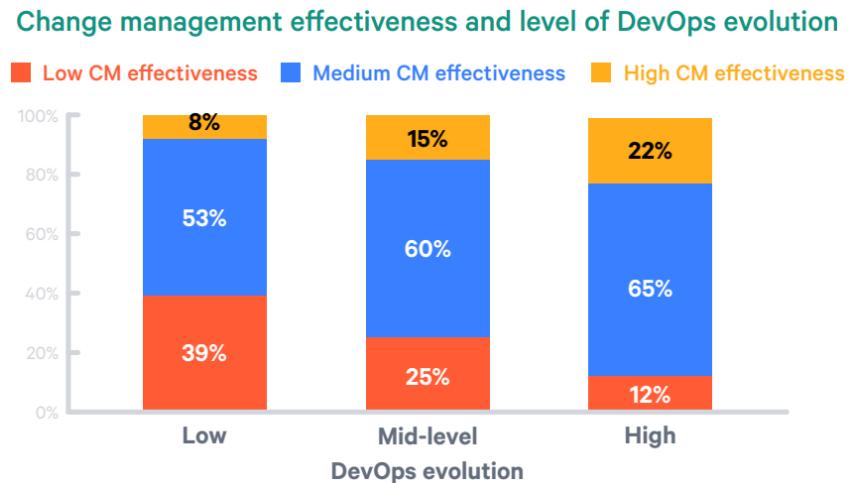


Figura 1.14: State of DevOps 2020 - Change Management

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, 1999, Extreme Programming Explained: Embrace Change

1.4.1 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*[17] 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.

$$\begin{array}{c}
 \text{Potential} \\
 \text{Revenue from} \\
 \text{Reinvestment} \\
 = \quad \text{Time Recovered} \\
 \text{and Reinvested} \\
 \text{in New Features} \\
 \times \quad \text{Revenue} \\
 \text{Generating} \\
 \text{Features}
 \end{array}$$

[WHERE]

Revenue Generating Features equals

$$\left(\begin{array}{l} \text{Frequency of} \\ \text{Experiments per} \\ \text{Line of Business} \end{array} \right) \times \left(\begin{array}{l} \text{Lines of} \\ \text{Business in the} \\ \text{Organization} \end{array} \right) \times \left(\begin{array}{l} \text{Idea} \\ \text{Success} \\ \text{Rate} \end{array} \right) \times \left(\begin{array}{l} \text{Idea} \\ \text{Impact} \end{array} \right) \times \left(\begin{array}{l} \text{Product} \\ \text{Business} \\ \text{Size} \end{array} \right)$$

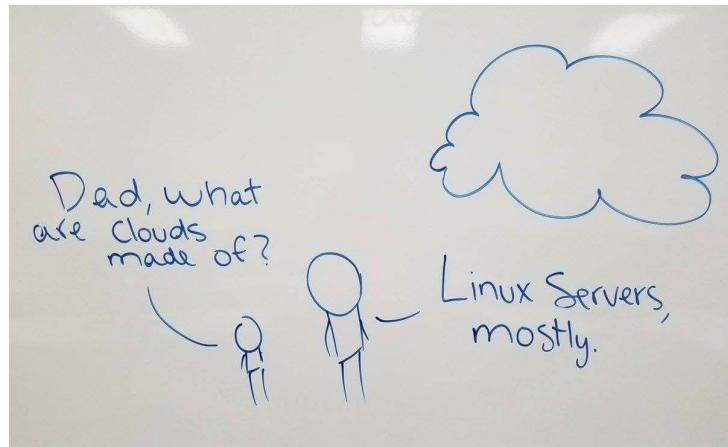
Figura 1.15: DevOps - Impatto sul ROI - Source: Veritis

Cosa si può dire riguardo l'impatto nel medio e lungo 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 2

Sviluppo ed Automazione su Cloud



2.1 Introduzione

Si sente spesso parlare in ambito informatico del concetto di **Cloud Computing**, ma il termine in se può avere diverse accezioni in base al tipo di "cloud" di cui parliamo. Il concetto generale vede il cloud come un nuovo paradigma per il quale le risorse informatiche vengono *localizzate* all'esterno della rete locale e vengono sfruttate come una **entità manipolabile e configurabile**, senza limiti di utilizzo o scalabilità.

In altre parole, per definire il *Cloud Computing* si necessita che:

- l'utente sia in grado di ottenere risorse informatiche *on-demand* senza l'interazione umana con i fornitori dei servizi;
- le risorse siano accessibili in modo libero da qualsiasi device o altra risorsa nella rete, che sia privata o pubblica;
- le risorse del fornitore siano fruibili da più utenze contemporaneamente, usando schemi multi-utente o *multi-tenant*, isolamento a livello logico o di rete, ed allocabili velocemente ed elasticamente (scalabili);
- i sistemi si auto-gestiscono, regolando la fruizione delle risorse, sfruttando sistemi di monitoraggio automatico basati su adeguati livelli di astrazione logica e fisica. L'utilizzo deve poter essere monitorato sia dal fornitore che dal fruitore delle risorse, in tempo reale.

Tipologie Seguendo la definizione di cloud del **NIST**[18], esistono diversi modelli di erogazione dei servizi:

- **Public Cloud:** risorse disponibili mediante internet pubblico, fruibili in modo gratuito (ma limitato) oppure a consumo (generalmente orario). I sistemi public cloud permettono una scalabilità totalmente trasparente all'utilizzatore e virtualmente infinita;
- **Private Cloud:** utilizzato quasi esclusivamente da imprese, e gestita generalmente dalle stesse, ubicando le risorse *on-premise* oppure in location esterne. E' concepita per andare incontro alle esigenze delle aziende che necessitano sicurezza e contenimento dei dati all'interno dell'ambiente aziendale stesso;
- **Hybrid Cloud:** unione delle prime due tipologie, generalmente nasce dalla partnership tra una azienda ed un provider di public cloud, così da poter mantenere parte delle risorse *on-premise* (più sensibili) e una parte sul public cloud, così da facilitare la fruibilità da parte di interni ed esterni.

The Cloud Pyramid

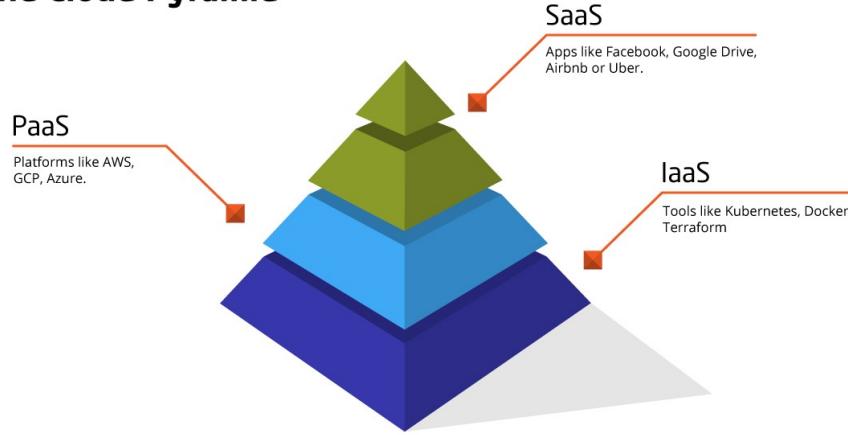


Figura 2.1: La Piramide del Cloud - **Source:** Vladimir Fedak

Modelli di Servizio Sempre il **NIST** definisce un modello noto come *SPI*:

- **Infrastructure-as-a-Service:** mette a disposizione macchine virtuali (VM), storage virtuale ed infrastrutture di rete virtuali che i fruitori possono creare e configurare a piacere. Il provider fornisce quindi una infrastruttura *bare metal* dove poi il cliente dovrà costruire il proprio pacchetto e sviluppare applicativi sul sistema operativo;
- **Platform-as-a-Service:** offre una versione semplificata di infrastruttura, oltre a servizi di sistema operativo, applicativi, framework di sviluppo e metodi di controllo degli stessi. Il cliente può così creare i propri applicativi unendo infrastrutture *bare metal* ad applicazioni gestite dalla piattaforma;
- **Software-as-a-Service:** ambiente puramente applicativo, gestito dal provider, con interfaccia utente o *API* sfruttabili dall'utente.

Tipica caratteristica di tutti i modelli di Cloud, è il fornire servizi secondo un **SLA** (*Service Level Agreement*)[19], ovvero un contratto dove si definiscono le metriche di qualità del servizio che devono essere rispettate dal fornitore nei confronti dei propri clienti. Generalmente il **SLA** si articola in definizione degli indicatori di qualità, realizzazione di un sistema di reporting, condivisione dei valori target e monitoraggio dei valori rilevati.

2.2 Infrastructure-as-Code

In un ambiente *Cloud*, dove le risorse sono definibili mediante *API* e semplici richieste al provider dei servizi, è nata una pratica volta ad unificare lo sviluppo del codice per la definizione di intere infrastrutture in ottica *DevOps*, permettendo così di sfruttare sia sviluppatori che operationals per gestire ciò che ospiterà l'applicativo sviluppato in un progetto.

L'*Infrastructure-as-Code*[20] è quindi il processo che gestisce delle infrastrutture mediante l'utilizzo di codice *machine-readable* al posto di configurazioni manuali o interfacce grafiche. Nel tempo le infrastrutture informatiche sono state oggetto di processi sempre manuali, gli addetti andavano fisicamente nel datacenter ad installare un nuovo server per poi configurarlo e renderlo fruibile a tutti.

Il primo problema che si andava a creare è certamente il **costo**, mantenere molte figure in grado di gestire una infrastruttura di un certo calibro fa lievitare i costi annuali notevolmente, oltre al grado aggiunto di gestione del personale che aggiunge un altro livello di overhead sui processi. I problemi successivi riguardano la **scalabilità** e **affidabilità** delle risorse installate, che necessitavano di continue revisioni e modifiche fisiche.



Figura 2.2: Infrastructure-as-Code - Source: HashRoot

L'avvento del *Cloud Computing* ha permesso di eliminare molti di questi problemi o di trovare un modo per mitigarli efficacemente, e l'**IaC** può essere definito come il "tassello mancante" a poter eliminare tutti quei processi manuali, spesso proni ad errori o a lavoro usa e getta. L'infrastruttura viene così definita a livello di codice, mediante l'uso di linguaggi diffusi o di configurazioni sempre versionabili, portando così diversi vantaggi ai processi:

- **Velocità:** creare una infrastruttura cloud diventa estremamente semplice anche per chi non ha skills puramente sistematiche, e diventa semplice replicarla su più ambienti (dev/stg/prod);
- **Consistenza:** se un processo manuale può portare ad errori e risultati inconsistenti, nell’IaC il codice è l’unica “*source of truth*”, che definisce tutto, dalle risorse al processo per crearle, modificarle e distruggerle;
- **Tracciabilità:** essendo semplice codice, può essere utilizzato un VCS per tenere traccia delle sue modifiche e nel caso effettuare dei rollback;
- **Efficienza:** il processo ne risente positivamente dall’uso di IaC, ad esempio per creare ambienti di testing sempre uguali e replicabili all’infinito anche da uno sviluppatore, e in fase di deployment finale sfruttare lo stesso codice per gli ambienti di produzione;
- **Costi:** l’utilizzo del cloud e dell’IaC permette di ridurre e tenere sott’occhio continuo i costi della propria infrastruttura, oltre a risparmiare sul personale addetto alla sua creazione e manutenzione.

L’approccio al codice può essere di tipo **imperativo** o **dichiarativo**, in base al tool scelto, dove con imperativo si intende il “come” devo modificare o creare le risorse, mentre nel dichiarativo si parte da uno stato desiderato delle cose ed il sistema deciderà le cose da fare in base allo stato attuale. Questo codice può inoltre essere integrato nei processi *DevOps-oriented*, permettendo di estendere l’automazione in modo che si riesca a creare anche l’infrastruttura oltre che deployare il software creato.

I principali strumenti e framework per l’*Infrastructure-as-Code* sono:

- HashiCorp Terraform (multicloud)
- AWS CloudFormation
- Microsoft Azure Resource Manager
- Google Cloud Deployment Manager

2.3 Configuration-as-Code

Se mediante l’*Infrastructure-as-Code* possiamo definire, creare e gestire le risorse infrastrutturali, una volta create queste andranno configurate e i propri applicativi installati e mantenuti. La pratica di *Configuration-as-Code*[21] permette di sfruttare lo stesso concetto visto precedentemente, applicandolo ad un diverso ambito, anche se strettamente correlato se non addirittura a volte sovrapponibile con l’IaC.

Questa pratica permette di eliminare ciò che veniva fatto post installazione di nuovi server fisici, ovvero **installazione, configurazione e pubblicazione degli applicativi** su di essi, o semplicemente la configurazione degli ambienti dove poi sarebbero stati installati futuri applicativi in sviluppo. Mediante la definizione di codice *machine-readable* e auto-documentante, possiamo creare un sistema di configurazione unificato e sempre adatto all’ambiente in cui si trova, grazie alla possibilità di analizzare i sistemi *as-is* e poi agire di conseguenza via codice.

I vantaggi della *IaC* si riflettono quindi anche sulla *CaC*, grazie all’uso di un sistema di VCS e al trattamento del codice come unica sorgente di verità, oltre che contenente lo stato desiderato dei sistemi. La *CaC* permette quindi di avere le seguenti feature in modo semplificato:

- Sviluppo di processi di build e deployment;
- Creazione di ambienti dipendenti da determinate configurazioni;
- Integrazione degli ambienti nei processi di Quality Assurance;
- Gestione di chiavi e *secrets* in modo controllato.

I principali strumenti e framework per il *Configuration-as-Code* sono:

- Red Hat Ansible
- Chef
- Puppet

Da notare come questi strumenti siano anche utilizzabili come *Infrastructure-as-Code*, potendo fare all'effettivo entrambi i compiti in modo efficiente, ma con approccio più orientato alla configurazione piuttosto che alla creazione.

2.4 *Containers ed ambienti controllati*

Nel mondo delle infrastrutture IT, ci si è sempre preoccupati di riuscire a creare un ambiente di sviluppo o deployment più consolidato possibile, in modo da evitare mancanze di dipendenze, librerie o configurazioni necessarie al corretto funzionamento di un determinato applicativo. L'unità di misura sempre utilizzata è la "*macchina*", che può essere un server fisico, oppure un server virtuale (VM), dove poi si dovevano installare e configurare tutte le dipendenze dell'applicativo da deployare.

Un **container**[22] è invece una unità più piccola, definibile come un ambiente software in grado di eseguire ed isolare dall'esterno l'esecuzione di uno o più processi, prendendo spunto dal container fisico utilizzato nella logistica, dove il contenuto da trasportare rimane al suo interno ma il mezzo di trasporto può cambiare frequentemente (camion, treni, navi) senza cambiamenti nel modo in cui viene gestita quell'unità.

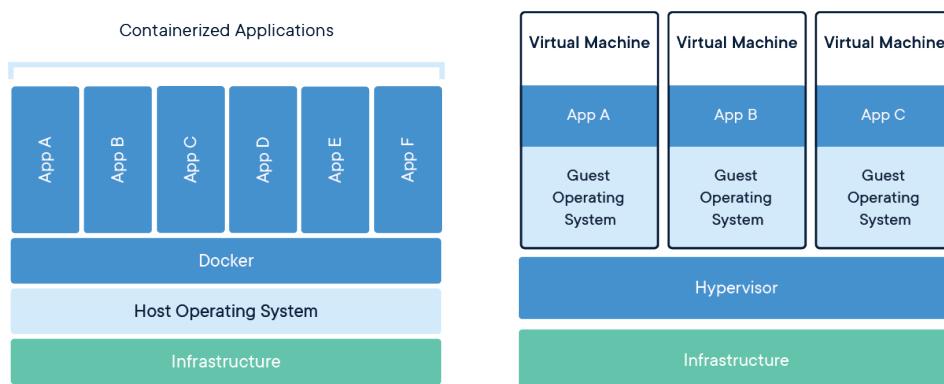


Figura 2.3: Dalle VM ai Container - **Source:** Docker

Allo stesso modo un container software risulta una unità facilmente trasferibile e deployabile in molti ambienti, *self-contained* ma che necessita di appoggiarsi ad un *runtime* che fornisca

una "base" su cui lavorare. Questo concetto non è decisamente nuovo, esistendo in modalità diverse fin dai primi anni 2000 su **FreeBSD** mediante le *jails*, ovvero divisioni dell'OS in parti indipendenti che però condividono lo stesso *kernel*. Queste implementazioni si sono poi evolute negli **LXC** (*Linux Containers*) nel 2008, la prima vera implementazione completa di un container in ambiente Linux, sopra il quale poi si sono diffusi diversi runtime di gestione quali *Warden* e *Docker*. Il concetto di container diventa "open" nel 2015 grazie alla nascita della **OCI** (*Open Container Initiative*), fondata da *Docker*, *CoreOS* e altre aziende del settore. Nel mentre anche in ambienti Windows, Microsoft implementa la possibilità di avere container nativi sui propri sistemi operativi.

L'utilizzo di containers in ambito Cloud permette di sviluppare applicativi *agnosticci* rispetto all'ambiente in cui vengono deployati, spostando così la creazione dell'ambiente corretto verso lo sviluppatore, che conosce le dipendenze necessarie al suo funzionamento. In un ambiente *DevOps-oriented*, dove il team è cross-funzionale, questi container possono essere utilizzati in modo estensivo in tutte le fasi del processo, dal testing alla delivery fino al deployment finale, permettendo di facilitare notevolmente tali fasi e di evitare il cosiddetto *vendor lock*, ovvero la forte dipendenza da una piattaforma o ambiente per utilizzare un determinato applicativo.

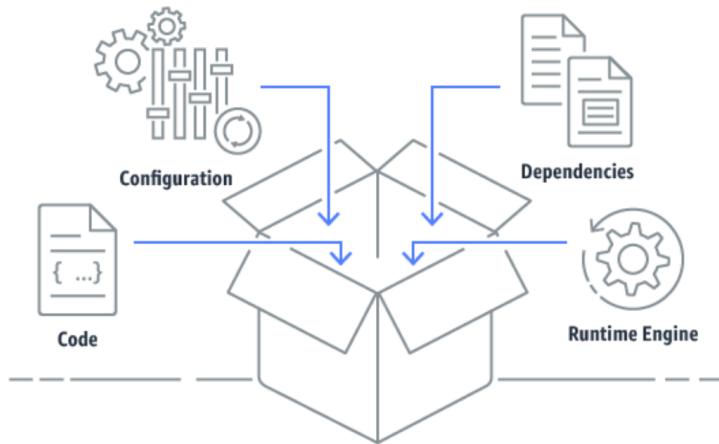


Figura 2.4: Contenuto di un Container - **Source:** AWS

La stessa cosa può inoltre essere applicata agli **ambienti di sviluppo locali**; infatti sulle singole macchine degli sviluppatori si possono gestire le dipendenze del progetto sfruttando sempre dei container, che possono essere quindi creati e distrutti a piacere per fare testing e build locali, senza preoccuparsi di avere un ambiente non in linea con il resto della catena produttiva. Un esempio potrebbe essere un applicativo basato sulla *Java Virtual Machine*, dove le dipendenze spesso non sono solo la macchina virtuale in se ma anche tutte le librerie accessorie; si può quindi pacchettizzare la JVM assieme al resto delle dipendenze in un singolo container, dove poi si potrà inserire l'applicativo pronto all'uso oppure sfruttarlo per creare build locali funzionanti.

I principali *container engine*[23] attualmente sul mercato sono:

- Docker RunC (basato su *libcontainer*)
- Red Hat crun (containers OCI)

Capitolo 3

Analisi del Processo di Sviluppo

3.1 Struttura del Progetto

Sphere si presenta come un sistema a *microservizi*, con interfaccia esterna mediante *API gRPC-based*, ed una user interface basata su app native per mobile (iOS al momento della scrittura).

Il codice è interamente contenuto in un **singolo repository** gestito su *Atlassian Bitbucket*, seguendo la filosofia del *monorepo*[24] proveniente dalle grandi aziende come Google, Microsoft e Facebook. Il monorepo permette di avere tutte le dipendenze in un singolo pacchetto, e di creare facilmente nuovi moduli al suo interno dipendenti da altri moduli o dipendenze già utilizzati. Ogni microservizio o modulo mobile è contenuto in una sottocartella del repository, che agisce come modulo dell'intero progetto, grazie all'utilizzo di un Build System unificato.

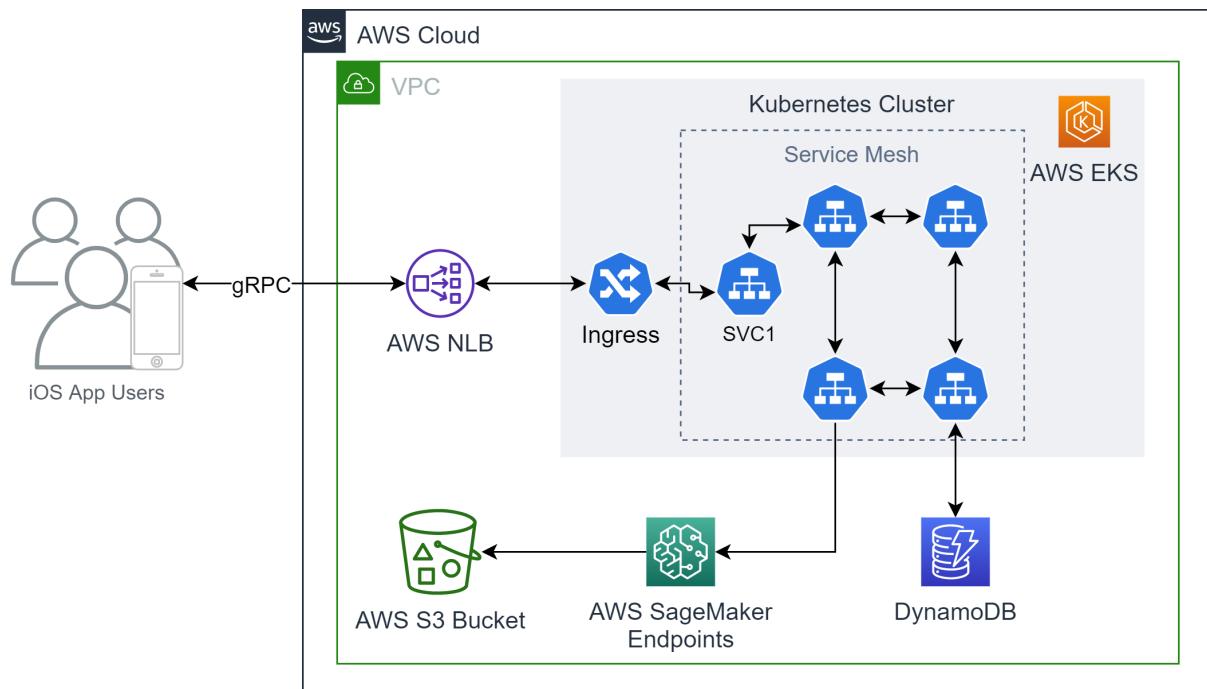


Figura 3.1: Sphere - Architettura

I microservizi si dividono in base al linguaggio su cui sono sviluppati, a scelta tra *C++* e *Python 3*, ognuno con le sue dipendenze come descritte nel build system, mentre i moduli

mobile si articolano in base alla loro feature, sono scritti interamente in *Swift* e necessitano di ambiente *macOS* per essere compilati. Le interfacce tra i microservizi o tra i moduli mobile e i microservizi sono basate su *gRPC* con formato *ProtoBuf*, mentre tutti i microservizi vengono compilati e pacchettizzati in *container Docker* così da poter seguire il deployment in ambiente *Kubernetes*.

3.2 Il processo di sviluppo

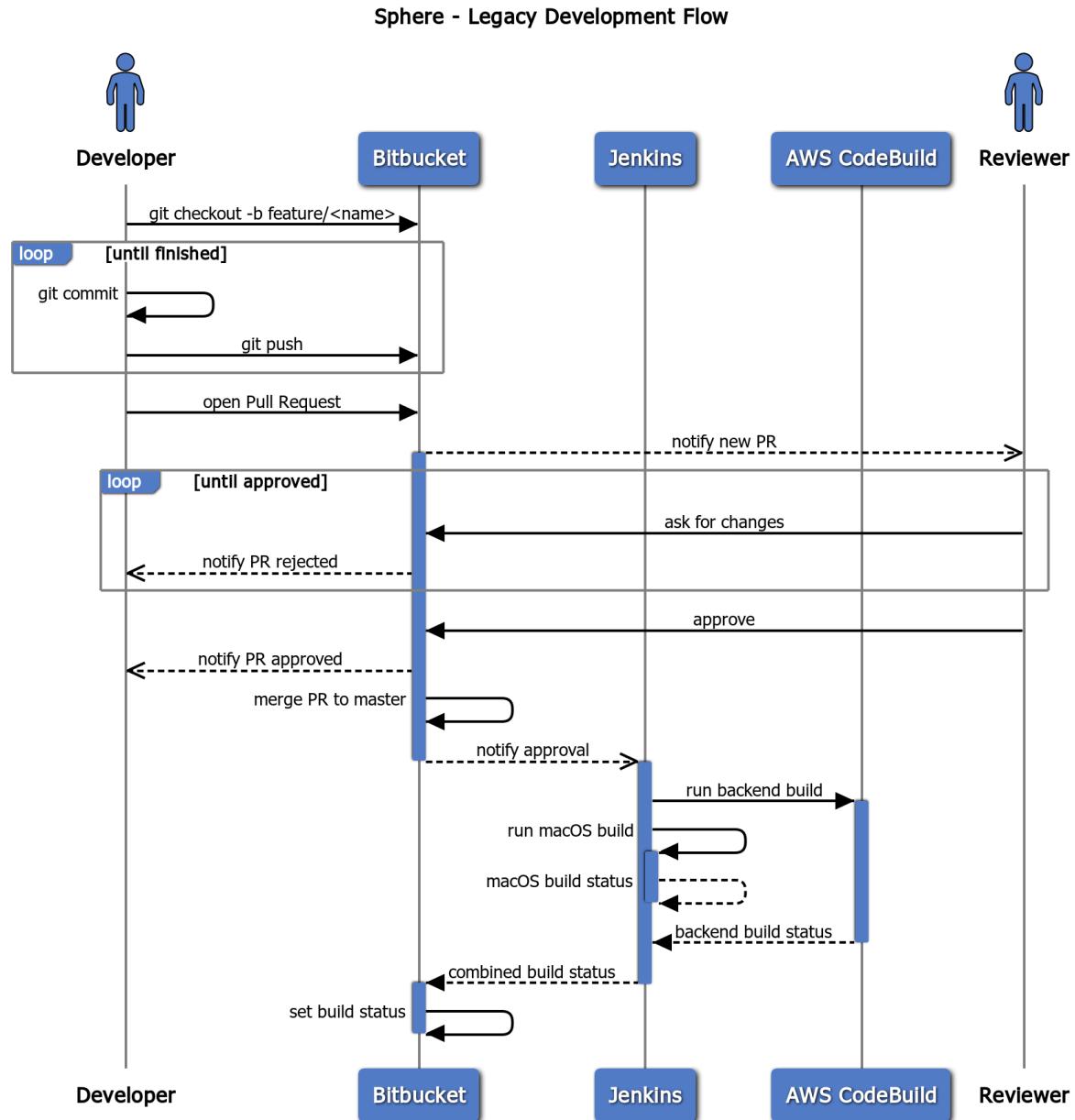


Figura 3.2: Sphere - Workflow Legacy

Il processo in atto prima dell'analisi *DevOps* (figura 3.2) si compone di una gestione a *feature branch*, dove il singolo sviluppatore tiene tutte le sue modifiche rispetto alla *master branch*, ed

una volta finito lo sviluppo richiede la code review mediante l'utilizzo di una Pull Request (PR) su Bitbucket.

L'apertura di una PR scatena una notifica verso il reviewer, senza però interpellare il sistema di *CI* (Jenkins), che viene scatenato solo in fase di accettazione e merge della PR sulla master branch. La *Declarative Pipeline* in atto effettua la build degli artefatti iOS direttamente sulla macchina master (Apple Mac Mini) mentre fa offloading della build dei servizi di backend utilizzando *AWS CodeBuild*, da notare come in nessun caso vengano analizzati i test (unit e integration) e un fallimento avvenga solo ed esclusivamente se la compilazione fallisce.

Una **parte mancante** nel processo è di sicuro la **delivery degli artefatti** prodotti, infatti ne i moduli per iOS né i servizi di backend vengono compilati e mantenuti in un repository da cui poi attingere in fase di deployment, che viene fatto interamente a mano a seguito di una build manuale, upload manuale e controllo degli artefatti (Docker Containers). Nella nostra analisi non ci concentreremo sulla delivery degli artefatti iOS, su cui è stata applicata una policy differente da parte del team competente.

Possiamo definire questo processo un primo passo verso un approccio *DevOps*, con diverse mancanze e punti deboli, che andremo a consolidare con l'analisi completa dei requisiti e la creazione di un processo *full DevOps compliant*.

3.3 Requisiti e KPI

A seguito della analisi del processo preesistente, sono stati formulati dei requisiti funzionali e associati a relativi KPI secondo l'approccio **SMART** (*Specific, Measurable, Attainable, Relevant, Time-bound*), che il nuovo processo dovrà rispettare per ottenere la qualità attesa e i maggiori benefici:

- **RVisibility:** il processo dovrà essere sempre disponibile alla consultazione e revisione, anche senza il supporto della infrastruttura on-premise;
 - **Cloud Readiness** (si/no): gli strumenti utilizzati devono essere disponibili online in ambienti cloud, per migliorare la fruibilità da parte di tutte le utenze;
- **RAvailability:** il processo dovrà mantenere un livello di *liveness* oltre il 90% del tempo di utilizzo complessivo, e poter scalare in caso di aumenti di carico;
 - **Average Availability** (ore per giorno): gli strumenti devono essere sempre disponibili durante tutte le ore di tutti i giorni della settimana, anche se necessitassero di un tempo di *ramp up* minimo;
- **REfficiency:** il processo deve mantenere un livello di efficienza elevato, riducendo al minimo i tempi morti e fornendo agli sviluppatori un sistema sempre pronto all'uso;
 - **Average Build Time** (in minuti): il processo di *Continuous Integration* deve mantenere un tempo medio di build e testing ragionevole (5 minuti), così da non rallentare l'analisi delle PR da parte dei reviewers;
 - **Average Build Count** (per giorno): numero di volte che il processo di integrazione/delivery viene utilizzato nell'arco di una giornata lavorativa media;

- **RReport**: il processo dovrà creare un report continuo con l'andamento dello stesso (build, test), per valutare la qualità del codice nel tempo;
 - **Report Available** (si/no): definisce se il processo ha implementato correttamente un sistema di reportistica per l'andamento delle build e dei test;
- **RQuality**: si dovrà implementare un sistema di code analysis per prevenire il *technical debt* in fasi avanzate di sviluppo;
 - **QA Available** (si/no): definisce se il processo ha implementato correttamente un sistema di code analysis per quality assurance del codice;
- **RTooling**: utilizzare un tool dedicato a gestire il processo fin dalle primissime fasi, al posto di lasciare l'accesso incondizionato ai repository;
 - **Unplanned Work Rate** (0-10): il processo deve ridurre al minimo la quantità di lavoro non pianificato, per la sua manutenzione o implementazione di nuove features al suo interno;
- **RCost**: l'infrastruttura a supporto del processo deve contenere i costi il più possibile rispetto al minimo delle risorse necessarie al suo corretto funzionamento;
 - **Process Cost** (in \$, per run): il costo del processo non deve eccedere il budget allocato aziendalmente.

3.4 Analisi del Processo Legacy

Prendendo in considerazione il processo descritto precedentemente, ed in atto al momento dell'inizio della fase di transizione verso il nuovo, rileviamo i seguenti dati:

Requirement	KPI Name	KPI Value	Achieved?
<i>RVisibility</i>	Cloud Readiness	✗	✗
<i>RAvailability</i>	Average Availability (per day)	8 hrs	✗
<i>REfficiency</i>	Average Build Time	19 mins	✗
	Average Build Count (per day)	5	✗
<i>RReport</i>	Report Available	✗	✗
<i>RQuality</i>	QA Available	✗	✗
<i>RTooling</i>	Unplanned Work Rate	4/10	✓
<i>RCost</i>	Process Cost (per run)	0.4\$	✓

Tabella 3.1: Requisiti e KPIs - Workflow Legacy

Tali dati evidenziano una marcata **lentezza in fase di compilazione**, con tempistiche molto elevate che costringono gli sviluppatori a notevoli tempi morti di attesa. Come conseguenza dell'utilizzo della CI solo in fase di merge, il **numero di compilazioni giornaliere è molto basso** e limitato solo al controllo finale nel codice in una fase molto tardiva dello sviluppo. Inoltre, essendo gestito solo ed esclusivamente da una macchina *on-premise*, eventuali cali di tensione, tagli di connessione ad internet o similari possono causare *downtime* inaspettati, necessitando quindi presenza di una persona fisica in ufficio nel caso sorgessero problemi.

Di pro troviamo un **rate di lavoro inaspettato relativamente basso**, non essendo stata effettuata particolare manutenzione o aggiornamenti agli strumenti usati, come nemmeno patch di sicurezza, mentre l'unico costo extra rispetto alle risorse on-premise risiede nei tempi di compilazione mediante AWS CodeBuild, che applica un **costo basso**.

3.5 Il processo DevOps

Per implementare un processo *DevOps* secondo i requisiti desiderati, si è analizzato il mercato in base alle *Best Practices* utilizzate in ambito enterprise, sondando gli strumenti più utilizzati e rispondenti alle necessità, ed implementando il tutto tenendo in mente un costo contenuto senza perdita di performance.

3.5.1 Architettura High-Level e Fasi

Il processo *high-level*, come descritto nell'immagine 3.3, si compone di 5 fasi principali:

1. **Sviluppo del Codice**: lo sviluppatore, basandosi sulla master branch, scrive il codice localmente fino ad arrivare ad uno stato voluto delle cose;
2. **Review del Codice**: lo sviluppatore apre una Pull Request (Code Review) mediante un sistema che orchestra il flusso di sviluppo;
3. **Continuous Integration**: l'orchestratore avvia il testing automatico integrando i cambiamenti nella codebase principale, una volta finito il testing il tool di CI restituisce lo stato dei test che verranno valutati dall'orchestratore. Se le modifiche sono valide, il revisore può decidere di accettare la Pull Request, in caso contrario lo sviluppatore verrà notificato dall'orchestratore del fallimento e sarà chiesto di modificare il codice;
4. **Accettazione**: il revisore accetta la Pull Request, viene notificato lo sviluppatore e l'orchestratore integra le modifiche nella master branch. Subito dopo, l'orchestratore richiama la *Continuous Integration* per avviare un secondo giro di testing ed analizzare il codice (statico e dinamico), ricavando le metriche qualitative;
5. **Continuous Delivery**: al momento della release, lo sviluppatore tagga la parte di software da rilasciare mediante una semantica definita, invia il tag al repository di codice che notificherà il processo di *Delivery*. Questo processo creerà gli artefatti software e li distribuirà a sua volta nel repository, seguendo una semantica derivata dal tag iniziale.

3.5.2 Gestione del Codice

Il progetto *Sphere*, come accennato nella introduzione del capitolo, si compone di un *monorepo*, ovvero tutti i suoi moduli, dipendenze e librerie vengono gestiti da un singolo repository diviso in sottocartelle e gestito da un build system strutturato.

Il repository viene hostato su **Atlassian Bitbucket** mediante backend **Git**, il sistema *de facto* standard per il code versioning. Nel contesto del flusso *DevOps*, Git viene sfruttato localmente per aprire *feature branches* senza la necessità di mantenerle anche in remoto, e viene sfruttato dall'orchestratore per integrare i cambiamenti effettuati nella codebase principale (master branch).

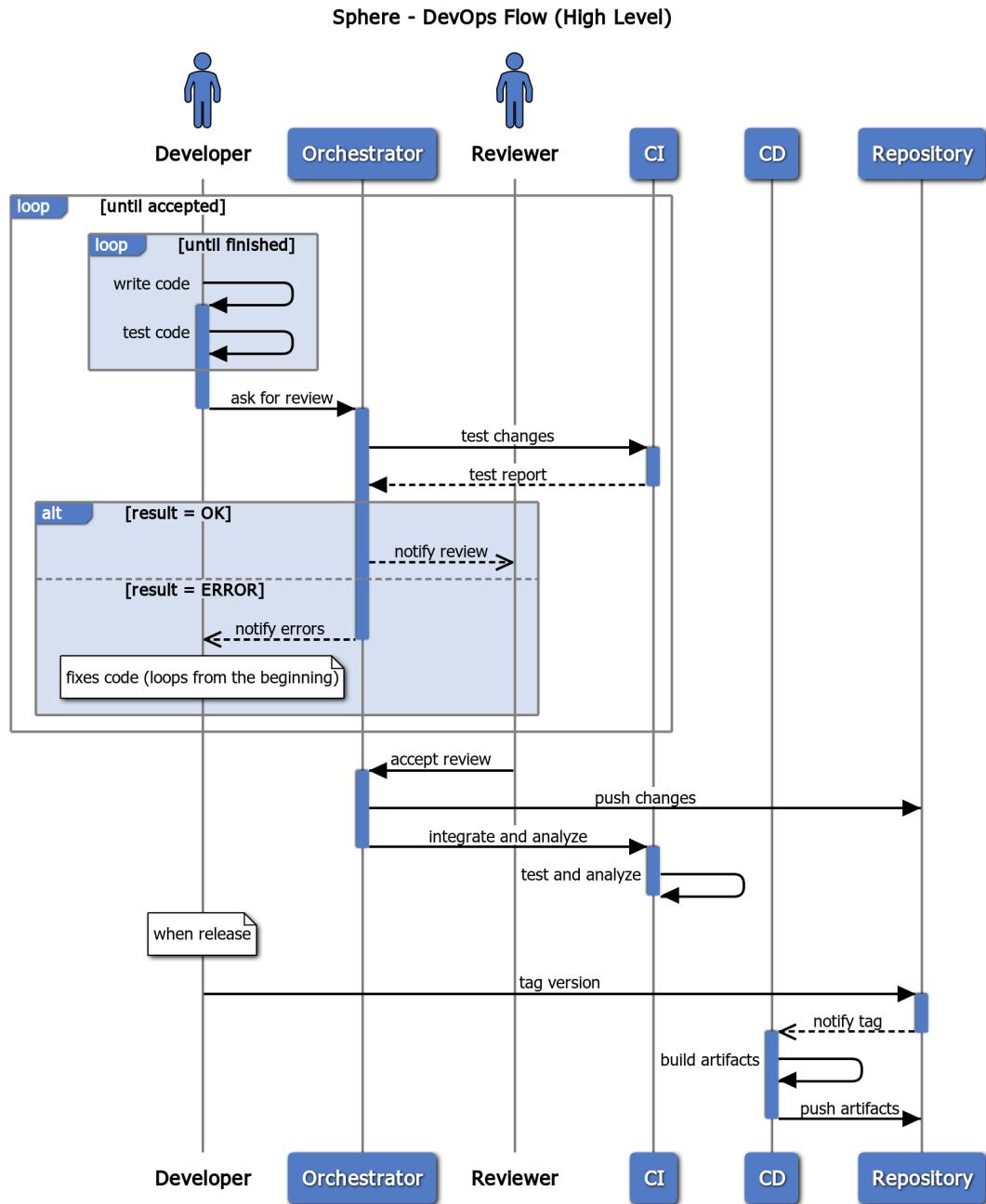


Figura 3.3: Sphere - Workflow DevOps

3.5.3 Pull Requests e Code Review

Uno dei capisaldi delle metodologie *Agile* e *DevOps* è la *inspection*, tra cui nasce la pratica che permette di ottenere codice di alta qualità grazie alla sua condivisione con il resto del team, che potrà così visionarlo, commentarlo e fornire feedback continuo sul suo sviluppo. Questo modo di lavorare viene denominato *Code Review*, spesso associata alla *Pull Request*, ovvero la richiesta di "merge" delle modifiche nella codebase principale previa approvazione da parte di uno o più revisori.

Esistono diversi metodi di Code Review, ma in questa casistica è stato scelto di utilizzare delle Pull Request assistite da uno strumento (l'orchestratore) per permettere a tutti i team di

lavorare in modo continuo grazie alla possibilità di poter visionare facilmente le modifiche di una PR, commentarle online collettivamente e fornire feedback sempre efficace.

3.5.4 Continuous Integration

La prima fase automatizzata del processo consiste nella pipeline di *Continuous Integration* (CI), gestita da uno strumento dedicato che funge da "motore" ed esegue tutte le sue fasi in base agli input dell'orchestratore. Il suo scopo primario risiede nel controllare tutte le modifiche potenzialmente applicabili dalla accettazione di una Pull Request, integrandole "temporaneamente" nella codebase principale in un ambiente controllato ed avviando il testing o l'analisi del codice in base alla fase del processo.

Test Automation

Il codice deve essere preparato e scritto in modo da integrare al meglio possibile dei test automatici, sia **Unit** che **Integration**, per valutare sia la funzionalità sviluppata che la sua interazione con il resto dei sistemi. Tali test vengono poi avviati dal motore della pipeline di *CI*, risultando in un report contenente lo stato di tutti i test eseguiti, analizzabile da uno strumento esterno o da un revisore.

Code Analysis

Oltre al testing, il codice può presentare dei potenziali bug, oppure dei code smell che possono sfociare in *technical debt* in un futuro prossimo. Per scongiurare queste cose, viene integrato un sistema di **analisi statica e dinamica** del codice, interpellato al momento dell'unione delle modifiche con la codebase principale, e che fornisce una valutazione di ciò che si può definire la "qualità" del codice. Le metriche per il quale viene valutata la qualità sono configurabili in base alle esigenze del progetto, così da non risultare troppo banali o troppo restrittive.

3.5.5 Continuous Delivery

L'ultima fase automatizzata del processo consiste nella pipeline di *Continuous Delivery*, gestita spesso dallo stesso motore della *CI*, permette di compilare, pacchettizzare e raccogliere tutti gli artefatti software pronti per una release da effettuare o per utilizzi futuri. Nel processo definito, questo processo viene utilizzato *on demand* in base alla pianificazione delle release, e richiamabile su singoli moduli del progetto.

Build Automation

Grazie alla integrazione della codebase con un *Build System* unificato, risulta molto semplice effettuare automazione della compilazione dei singoli artefatti, in base ad uno specifico **target** definito in fase di richiesta della delivery. La compilazione avviene inoltre in un ambiente controllato e definito a priori, sempre uguale e riproducibili, riducendo così eventuali variabili in gioco che possono compromettere la qualità del prodotto finale.

Artifacts Delivery

Gli artefatti software prodotti mediante compilazione vengono poi pacchettizzati come *Containers* ed inviati ad un repository (registry) per storage ed utilizzi futuri in un ipotetico deploy in un ambiente scelto.

3.5.6 Deployment

Dalla descrizione del processo high-level si è esclusa l’analisi della fase di Deployment, essendo una questione legata molto al modello di business attuale, che non vede l’azienda come fornitore del servizio in se ma solo fornitore della tecnologia. Vi sono quindi gli ambienti di testing gestiti internamente, il cui deployment viene effettuato **manualmente** in base alle esigenze, facilitato dall’utilizzo di tecnologie come *Container Orchestration* mediante *Kubernetes*.

3.6 Tecnologie e Strumenti

3.6.1 SCM: Git

Originariamente sviluppato da *Linus Torvalds*, *Git*[25] si propone come un tool estremamente performante con la caratteristica di essere distribuito e non centralizzato, quindi ogni sviluppatore può mantenere una copia del codice localmente e lavorare senza necessità di essere collegato ad un repository centrale. La sua struttura permette di gestire efficacemente flussi complessi e strutture dati complesse, oltre che permettere lo storage di grandi file utilizzando **Git-LFS** (*Large File Storage*). Git è stato scelto inoltre per la sua compatibilità con Atlassian Bitbucket, e per il suo utilizzo estremamente diffuso.

3.6.2 Build System: Bazel

Progetto nato dal tool interno di *Google* chiamato *Blaze*, **Bazel**[26] è un build system che supporta progetti di grandi dimensioni in molteplici linguaggi, fornisce un linguaggio high-level per descrivere gli artefatti, le configurazioni e le dipendenze dei moduli da compilare, ed è multipiattaforma.

Bazel mantiene una **cache interna** di tutti gli artefatti compilati precedentemente, ricompilando solo quelli che necessitano di cambiamenti e velocizzando i tempi notevolmente, permette inoltre di utilizzare una cache remota condivisa tra i vari sistemi dove viene eseguito. Ad ogni sua esecuzione, leggendo i file descrittori di compilazione, crea un **Action Graph** che definisce tutte le azioni da eseguire e le dipendenze tra i vari moduli o artefatti, permettendo build sempre corrette, ermetiche e riproducibili.

3.6.3 Cloud Provider: AWS

Amazon Web Services (abbreviato AWS) è una sussidiaria del colosso dell’ecommerce *Amazon*, e si propone come il più grande *cloud provider* attualmente sul mercato con oltre 46 miliardi di dollari di fatturato annuo in costante crescita. I servizi proposti spaziano dall’*IaaS*, al *PaaS* fino al *SaaS*, mantenendo però forte vantaggio in ambito infrastruttura e servizi managed. AWS è stato scelto a livello progettuale per la sua potenza, flessibilità e competitività sul mercato.

3.6.4 Orchestrator: Phabricator ed Arcanist

Nato da un progetto interno a *Facebook*, il suo lead developer Evan Priestley ha poi lasciato l’azienda per continuare il suo sviluppo esternamente fondando *Phacility*, **Phabricator**[27] è un multi-tool di *development collaboration* che integra diversi strumenti utili in un meccanismo di sviluppo *Agile-oriented*.

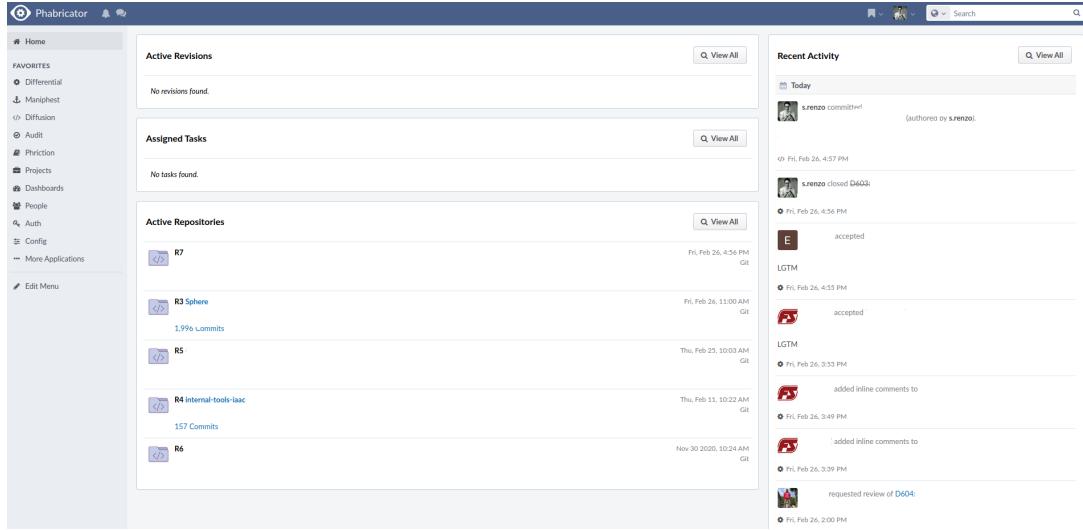


Figura 3.4: Phabricator Web UI

Nel progetto verrà sfruttato *Differential*, il sistema di Pull Requests e Code Review, e *Diffusion* che permette di integrare lo strumento con repository Git (anche remoti). Verrà inoltre descritto l’uso di **Arcanist**, strumento di *CLI* (Command Line Interface) che semplifica l’interazione con Phabricator e l’implementazione del flusso di sviluppo voluto.

3.6.5 CI/CD: Jenkins

Come sistema di automazione *CI/CD* è stato scelto **Jenkins**[28], software basato su Java in ambiente Apache Tomcat, *de facto* uno dei tool più utilizzati al mondo in fatto di pipeline di automazione, e che offre una interfaccia web semplice da utilizzare, illustrata nell’immagine 3.5.

Jenkins trova la sua forza nella flessibilità di utilizzo ed estensibilità, mediante vari tipi di oggetti creabili e gestibili e l’installazione di una miriade di *plugins* mantenuti sia ufficialmente che dalla comunità open source. Lo strumento permette inoltre di fare *offloading* di alcuni task verso macchine remote gestite dallo stesso, mediante il sistema degli **agents**.

3.6.6 Code Analysis: SonarQube

Il sistema di *static e dynamic code analysis* scelto è **SonarQube**, strumento molto diffuso e open source, che permette di analizzare, quantificare e fare reporting di metriche di qualità del codice in base a regole e soglie ben definite da un "profilo" di qualità, e che fornisce una web UI per visionare tali analisi, illustrata in figura 3.6.

SonarQube permette di analizzare potenziali bug, code smells, codice duplicato o poco testato, con delle semplici configurazioni e tool di supporto come **Sonar Scanner** che interagiscono col sistema centrale facilitando le analisi.

The screenshot shows the Jenkins dashboard. On the left, there's a sidebar with links like 'Nuovo Elemento', 'Personne', 'Cronologia compilazioni', 'Relazione progetto', 'Controlla impronta digitale file', 'Gestisci Jenkins', 'Le mie viste', 'Disk Usage', 'Open Blue Ocean', 'Lockable Resources', and 'Nuova vista'. The main area has a table titled 'Tutto' showing builds for 'Sphere', 'Sphere-CD', and 'Sphere-CD-Trigger'. The table includes columns for 'S' (Status), 'A' (Last build), 'Nome' (Name), 'Ultima compilazione completata con successo' (Last successful build), 'Ultima compilazione non riuscita' (Last failed build), 'Ultima durata' (Last duration), 'Built On' (Built On), 'Rev' (Revision), and '# Issues' (Number of issues). Below the table, there are links for 'Atom feed per tutte le compilazioni', 'Atom feed per le compilazioni non riuscite', and 'Atom feed solo per le ultime compilazioni'. At the bottom right, it says 'REST API' and 'Jenkins 2.263.4'.

Figura 3.5: Jenkins Web UI

The screenshot shows the SonarQube dashboard. On the left, there are filters for 'Quality Gate' (Passed: 8, Failed: 0), 'Reliability' (Bugs: A: 8, B: 0, C: 0, D: 0, E: 0), 'Security' (Vulnerabilities: A: 8, B: 0, C: 0, D: 0, E: 0), 'Maintainability' (Code Smells: A: 8, B: 0, C: 0, D: 0, E: 0), and 'Coverage' (≥ 80%: 0, 70% - 80%: 0, 50% - 70%: 0, 30% - 50%: 0, < 30%: 8). The main area displays four analysis results for the 'Sphere' project. Each result includes a star icon, the project name, a 'Passed' status badge, the date of the last analysis, the number of bugs, vulnerabilities, code smells, and duplication, and the programming language (Python, Kotlin, C++, C++ (Community)).

Figura 3.6: SonarQube Web UI

3.7 Analisi del Processo *DevOps*

A seguito della descrizione, implementazione ed utilizzo del processo *DevOps* high level, si necessita la valutazione dei vari aspetti come descritti dai Requisiti e KPI:

Requirement	KPI Name	KPI Value	Achieved?	Gain
<i>RVisibility</i>	Cloud Readiness	✓	✓	+100%
<i>RAvailability</i>	Average Availability (per day)	24 hrs	✓	+67%
<i>REfficiency</i>	Average Build Time	4 mins	✓	-375%
	Average Build Count (per day)	15	✓	+67%
<i>RReport</i>	Report Available	✓	✓	+100%
<i>RQuality</i>	QA Available	✓	✓	+100%
<i>RTooling</i>	Unplanned Work Rate	2/10	✓	-50%
<i>RCost</i>	Process Cost (per run)	2.6\$	✓	+84%

Tabella 3.2: Requisiti e KPIs - Workflow DevOps

Possiamo notare il notevole **guadagno** in merito di **tempo di compilazione** e **numero di compilazioni** giornaliere, sintomo che il processo implementato permette di ridurre i tempi morti degli sviluppatori ed ottimizzare al meglio il *throughput* massimo. Il processo aggiunge inoltre un sistema di *report* visibile a tutti, gestito dagli strumenti in base al loro scopo, assieme ad un tool di *Quality Assurance* del codice (code analysis) che permette di ridurre il debito tecnico nel tempo se affrontato nel modo corretto. Altro punto a favore è sicuramente la **cloud readiness** dell'intero deployment, che permette di essere ricreato in pochissimo tempo e riprodotto all'infinito.

L'unico reale punto negativo, seguendo le metriche definite, è il **costo** del processo per ogni volta che viene sfruttato. Questo deriva dalle **risorse Cloud** necessarie a mantenere attivi gli strumenti scelti e sempre disponibili da parte degli sviluppatori, oltre che dall'accesso alle risorse mediante rete protetta (VPN). Ovviamente questa nuova infrastruttura permette di ottenere una manutenzione meno indispensabile, se non con mirati aggiornamenti di sicurezza o *features* ove necessarie, sempre effettuate grazie all'utilizzo dei tool di *IaC* e *CaC*.

Capitolo 4

Tecnologie di Background

4.1 Cloud Provider: AWS



Figura 4.1: Logo Amazon Web Services - **Source:** AWS

Per implementare il processo high-level descritto nel capitolo precedente, ci si è forniti del provider di servizi cloud *Amazon Web Services*, scelto per la sua completezza di servizi offerti e competitività sul mercato.

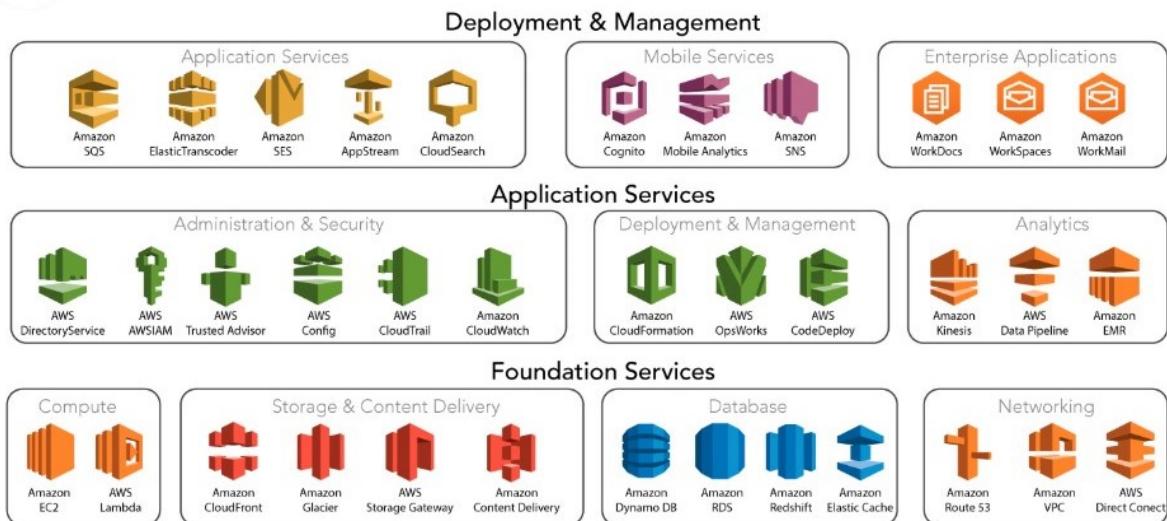


Figura 4.2: Servizi AWS (principali) - **Source:** AWS

I servizi che offre AWS (figura 4.2) si dividono sui 3 livelli della piramide cloud:

- **Infrastructure-as-a-Service**

- *Networking:* Virtual Private Cloud (VPC), Route 53 (DNS);

- *VM*: Elastic Compute Cloud (EC2);
- *Storage*: Elastic Block Storage (EBS);
- **Platform-as-a-Service**:
 - *Database*: DynamoDB, Relational Database Service (RDS);
 - *Mobile*: Cognito, Mobile Analytics;
 - *Messaging*: Simple Queue Service (SQS), Simple Notification Service (SNS), Simple Email Service (SES);
- **Software-as-a-Service**:
 - *Identity*: Identity and Access Management (IAM), Directory Service (DS);
 - *Monitoring*: CloudWatch, CloudTrail, X-Ray;
 - *Storage*: CloudFront, Simple Storage Service (S3).

Esistono molti altri servizi che non descriveremo ne utilizzeremo in questo progetto, questo fa intuire la vastità dell'offerta di AWS e come possa risultare estremamente flessibile in base alle esigenze. Nel nostro caso utilizzeremo i seguenti servizi per configurare la piattaforma che gestirà il processo *DevOps*:

- Networking basato su **VPC** e **Route 53** (reti private e DNS privato gestito);
- Macchine Virtuali basate su **EC2** con storage **EBS** di tipologia GP2 (General Purpose);
- Servizio VPN basato su **AWS Client VPN** (OpenVPN Server) per accedere alle risorse private;
- Load Balancing dei servizi basato su **Elastic Load Balancer** (ELB), usato in varianti *Classic* e *Application* in base all'applicativo;
- Docker Registry basato su **Elastic Container Registry** (ECR).

4.2 Infrastructure-as-Code: *Terraform*

Per creare l'infrastruttura sfruttando il paradigma di *Infrastructure-as-Code* è stato scelto l'utilizzo di **Terraform**[29], il tool *de facto* standard in ambienti multicloud che vogliono evitare il *vendor lock* delle proprie risorse.

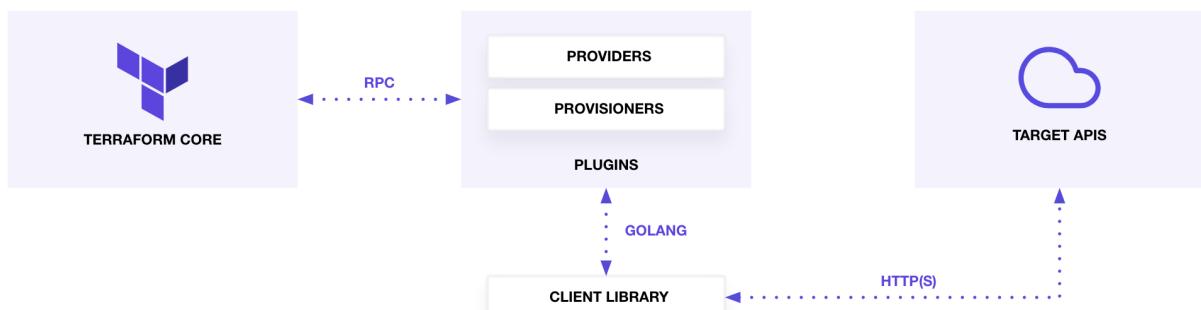


Figura 4.3: Terraform - Schema Funzionamento - **Source:** HashiCorp

Sviluppato da **HashiCorp** in *Go*, si presenta come una *CLI* ed un linguaggio dedicato chiamato **HCL** (*HashiCorp Configuration Language*), mediante il quale si esprime lo stato voluto delle risorse (dichiarativo) che poi il tool convertirà in azioni di creazione, modifica o cancellazione nell'ambiente cloud scelto.

Grazie alla sua struttura è possibile modularizzare e parametrizzare il codice, in modo da adattarsi a vari tipi di deployment e generalizzare le implementazioni, esattamente come si farebbe con un linguaggio comune (imperativo, funzionale, OO). Terraform, al momento di applicare le modifiche, **interroga** prima la cloud platform target sullo stato attuale delle risorse, adattando così il suo stato interno (che può essere mantenuto anche in remoto) e creando la **lista delle modifiche** da applicare per arrivare allo stato desiderato. Questa tecnica inoltre rileva e tiene conto delle **dipendenze tra le risorse** mediante la creazione di un grafo dedicato, ad esempio un entry DNS necessiterà di un target specifico, che dovrà essere quindi creato prima della entry stessa.

Un lato negativo nell'utilizzo di Terraform è la **non possibilità di effettuare dei rollback** mirati una volta che le modifiche sono state applicate. Per fare ciò servirà infatti fare un "revert" del codice modificato, e poi applicare di nuovo le modifiche, ma non vi è una funzione dedicata a riguardo.

4.2.1 Esempio: Creazione di una VM

Prenderemo come esempio la creazione di una istanza server virtuale in ambiente **AWS**, quindi la risorsa sarà di tipo **EC2** e necessiterà di diversi elementi di contorno per essere definita. In primis Terraform necessita di definire un **provider**, ovvero un "driver" che gli permetta di interfacciarsi con una cloud platform, in questo caso AWS, in un file nominato `main.tf`:

```
1 provider "aws" {
2     region = "eu-west-1"
3 }
```

Successivamente definiremo subito sotto la nostra istanza sfruttando la tipologia di risorsa chiamata `aws_instance` come descritta nella documentazione ufficiale:

```
1 resource "aws_instance" "example" {
2     ami           = "ami-something"
3     instance_type = "t3.micro"
4 }
```

In seguito bisognerà inizializzare Terraform mediante il comando `init` che permette di creare il "backend" (lo stato) del sistema e di scaricare tutte le dipendenze di cui necessita (providers, definizioni):

```
1 $ terraform init
2
3 Initializing the backend...
4
5 Initializing provider plugins...
6 - Checking for available provider plugins...
7 - Downloading plugin for provider "aws"
8
9 (...)
```

```

10
11 * provider.aws: version = "~> 2.10"
12
13 Terraform has been successfully initialized!

```

In seguito si potranno visualizzare ed applicare le modifiche usando i comandi `plan` ed `apply`, il primo della quale crea un report delle modifiche da applicare, mentre il secondo oltre a crearlo permette anche di applicare le modifiche listate in quel report:

```

1 $ terraform apply
2
3 (...)
4
5 Terraform will perform the following actions:
6   # aws_instance.example will be created
7   + resource "aws_instance" "example" {
8       + ami                               = "ami-something"
9       + arn                             = (known after apply)
10      + associate_public_ip_address     = (known after apply)
11      + availability_zone             = (known after apply)
12      + cpu_core_count                = (known after apply)
13      + cpu_threads_per_core          = (known after apply)
14      + get_password_data            = false
15      + host_id                      = (known after apply)
16      + id                            = (known after apply)
17      + instance_state               = (known after apply)
18      + instance_type                = "t3.micro"
19      + ipv6_address_count           = (known after apply)
20      + ipv6_addresses               = (known after apply)
21      + key_name                     = (known after apply)
22      (...)
23  }
24
25 Plan: 1 to add, 0 to change, 0 to destroy.
26
27 Do you want to perform these actions?
28 Terraform will perform the actions described above.
29 Only 'yes' will be accepted to approve.
30
31 Enter a value:

```

4.3 Configuration-as-Code: Ansible

Per la gestione della configurazione dei servizi all'interno dell'infrastruttura creata, è stato scelto l'utilizzo di **Red Hat Ansible**[30], uno dei tool di configuration più utilizzati al mondo assieme ai competitor *Chef* e *Puppet*.

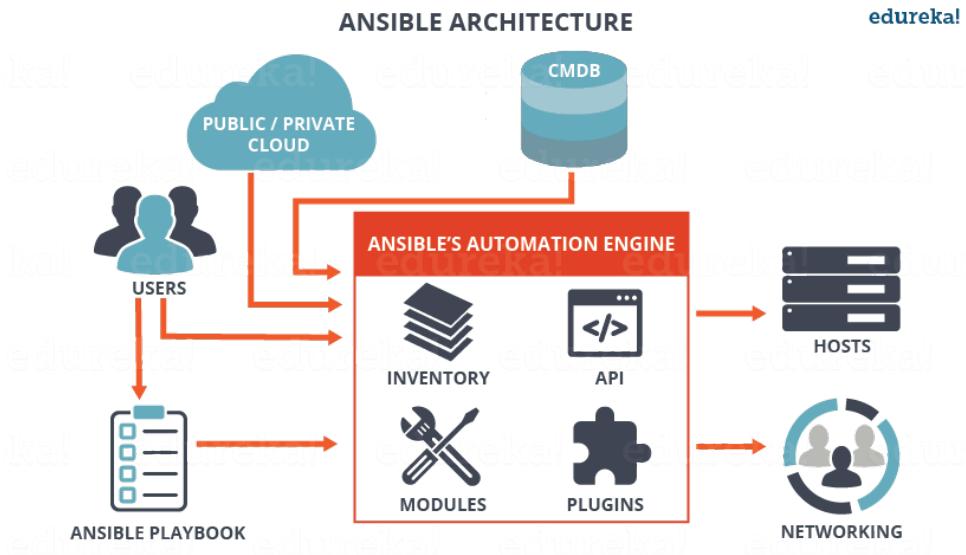


Figura 4.4: Ansible - Schema Funzionamento - Source: Edureka!

Ansible permette di gestire intere flotte di macchine mantenendo sempre lo stato delle cose ed uno stato desiderato, soppiantando così la vecchia pratica degli script "usa e getta" utilizzati in ambito sistemistico. Ansible sfrutta una sintassi semplice scritta in **YAML** (*YAML Ain't a Markup Language*) mediante ciò che viene definito un **playbook**, che definisce la lista ordinata dei task da effettuare per eseguire una determinata azione.

Mediante la creazione di **roles** si possono generalizzare i *playbook* creati, che possono poi essere eseguiti ad un livello superiore di astrazione. Per gestire le variabili di processo interne Ansible sfrutta i **facts**, richiamabili da qualsiasi punto dei *playbook* e dove si possono memorizzare dati anche lungo il corso di un processo. Infine, per definire i suoi target di lavoro (macchine locali o remote), viene creato un **inventory** in modo statico o dinamico, dove si descrive il target (IP, porta) e il metodo con cui collegarsi ad esso.

4.3.1 Esempio: installazione di Apache Httpd

Per definire un esempio di utilizzo degli *Ansible Playbooks*, illustreremo l'installazione del server Apache Httpd su un ipotetico host remoto mediante accesso SSH. In primis si necessita definire il file dell'*inventory* di Ansible, contenente il nome degli host (friendly), il loro IP e la porta a cui connettersi:

```

1 [group1]
2 host1 ansible_host=192.168.100.2 ansible_ssh_port=22
3 [group2]
4 host2 ansible_host=192.168.100.3 ansible_ssh_port=22

```

In seguito si scrive il *playbook* mediante codice **YAML**, definendo le azioni da fare in base al nome dato agli host nell'*inventory*. Tali azioni sono definite dal framework di Ansible stesso o mediante plugins esterni, possono definire condizioni, variabili e possono essere applicate più volte senza preoccuparsi di conflitti (Ansible controlla preventivamente lo stato):

```

1 ---
2 - hosts: group1 , group2
  tasks:

```

```

4   - name: Install Apache Httpd
5     yum:
6       name: httpd
7       state: present
8     when: ansible_system_vendor == 'AWS' and ansible_os_family
9       == 'RedHat'

```

Per avviare un Ansible Playbook basterà richiamare la CLI mediante un terminale (shell) con la seguente sintassi:

```
1 $ ansible-playbook -i hosts.ini playbook.yaml
```

dove `-i` indica l'*inventory* creato precedentemente e `playbook.yaml` il file di *playbook* creato con le nostre azioni da eseguire. Durante il processo potremo visionare le singole azioni che vengono compiute sui singoli hosts definiti, e otterremo alla fine un report con eventuali errori avvenuti durante il processo.

4.4 Container Engine: *Docker*

Come precedentemente descritto nel capitolo dedicato al Cloud, in un processo *DevOps* risulta importante mantenere consistente l'ambiente di sviluppo, testing e deployment del software. Per fare ciò vengono sfruttati i **Containers** e più precisamente l'engine di **Docker**[31], uno dei più diffusi al mondo e semplici da utilizzare.

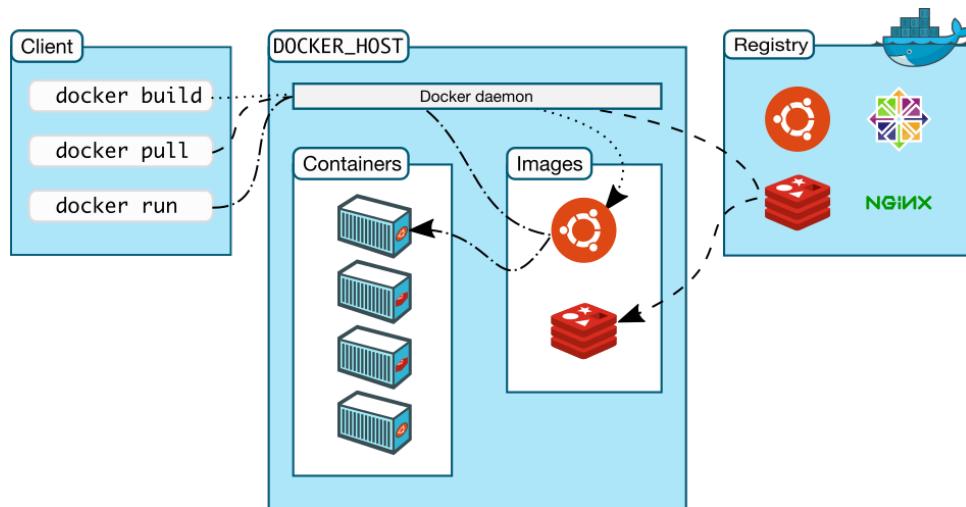


Figura 4.5: Docker - Schema Funzionamento - Source: Docker

Docker si compone di un **daemon** (processo) che gestisce il *container runtime* sottostante (basato su *runC*) e che offre un accesso facilitato ai container mediante delle API, ed un **client** usabile via CLI oppure interfaccia grafica (in base alle implementazioni). Oltre a gestire il runtime, Docker fornisce facile accesso alla gestione delle **immagini** dei container stessi, scaricabili online o compilabili localmente a partire da un **Dockerfile**, e fornisce inoltre un livello di astrazione sulla gestione dei **volumi** agganciabili ai container in esecuzione (mediante *binding* con una cartella locale, o mediante volumi interamente gestiti).

Oltre all'utilizzo per distribuire gli artefatti applicativi, nel progetto verrà utilizzato Docker per eseguire l'installazione e la configurazione facilitata degli strumenti da utilizzare sulle

macchine virtuali, evitando tediosi processi di installazione e velocizzando futuri aggiornamenti grazie alla mera sostituzione della immagine di base.

4.4.1 Esempio: applicazione Java

Avendo un applicativo Java sotto forma di file `.jar` eseguibile, possiamo definire il `Dockerfile` che contenga il runtime della JVM, le dipendenze necessarie e che permetta di eseguire l'applicazione mediante l'uso di un container engine:

```
1 FROM openjdk:8-jdk-alpine # Base image
2 WORKDIR /usr/app # Working directory
3 COPY ./app.jar app.jar # Copy file from local to container
4 ENTRYPOINT ["java", "-jar", "/usr/app/app.jar"] # Run the app
```

Mediante questo Dockerfile, a seguito della build con la CLI di `docker` e i comandi `build` e `tag`, verrà creata una immagine basata sulla immagine pubblica `openjdk:8-jdk-alpine` con il file `app.jar` al suo interno che verrà avviato come processo `java "controllato"` dal container al momento della sua creazione.

Capitolo 5

Architettura Cloud

Per poter sfruttare il processo *DevOps* descritto in precedenza, si necessita della creazione di una infrastruttura cloud che possa ospitare e gestire gli strumenti, i flussi, i dati e le utenze necessarie al processo stesso. In questo capitolo verrà descritta l'architettura in modo più specifico, come sono state definite le risorse e configurati gli strumenti utilizzati.

5.1 Introduzione e Definizione

Ogni strumento introdotto nel processo, necessita di un set di risorse fornite dal *cloud provider* in misura tale da riuscire a gestire le sue operazioni efficacemente ma al contempo ridurre i costi dove possibile. Nel nostro caso, ogni strumento necessita obbligatoriamente di una **istanza server** dove essere *hostato* singolarmente, ovvero ognuno ha una sua macchina dedicata e non condivide ambiente e dati con gli altri, un **metodo di accesso** ad esso in modo sicuro ed affidabile (possibilmente HTTPS) e la possibilità di non esporre su internet tali istanze così da ridurre al minimo possibilità di attacchi informatici a causa di vulnerabilità non corrette.

A seguito della analisi dei requisiti necessari, sono emerse le seguenti necessità a livello di risorse cloud da utilizzare:

- Le risorse devono essere raccolte in una rete virtuale **AWS VPC** per interconnetterle senza sforzi ulteriori;
- Ogni istanza sarà definita con una macchina virtuale (VM) su **AWS EC2**;
- Per accedere allo strumento installato sull'istanza si dovrà agganciare un **Load Balancer** (di tipo variabile in base alle richieste), sui cui verrà installato il certificato per l'utilizzo in **HTTPS** e che fornirà *TLS Termination* semplificando il setup della rete dietro di esso;
- Se lo strumento necessita di un database relazionale (Phabricator, SonarQube) verrà creata una istanza dell'engine necessario utilizzando **AWS RDS** nella stessa rete creata;
- Per permettere l'accesso sicuro e controllato alle risorse in cloud, si utilizzerà una istanza di **AWS Client VPN** con relativi certificati da rilasciare ai client abilitati.

Seguendo queste linee guida si è così definita una architettura cloud complessiva che potesse rispondere con successo alle esigenze del processo.

5.2 Diagramma Architetturale

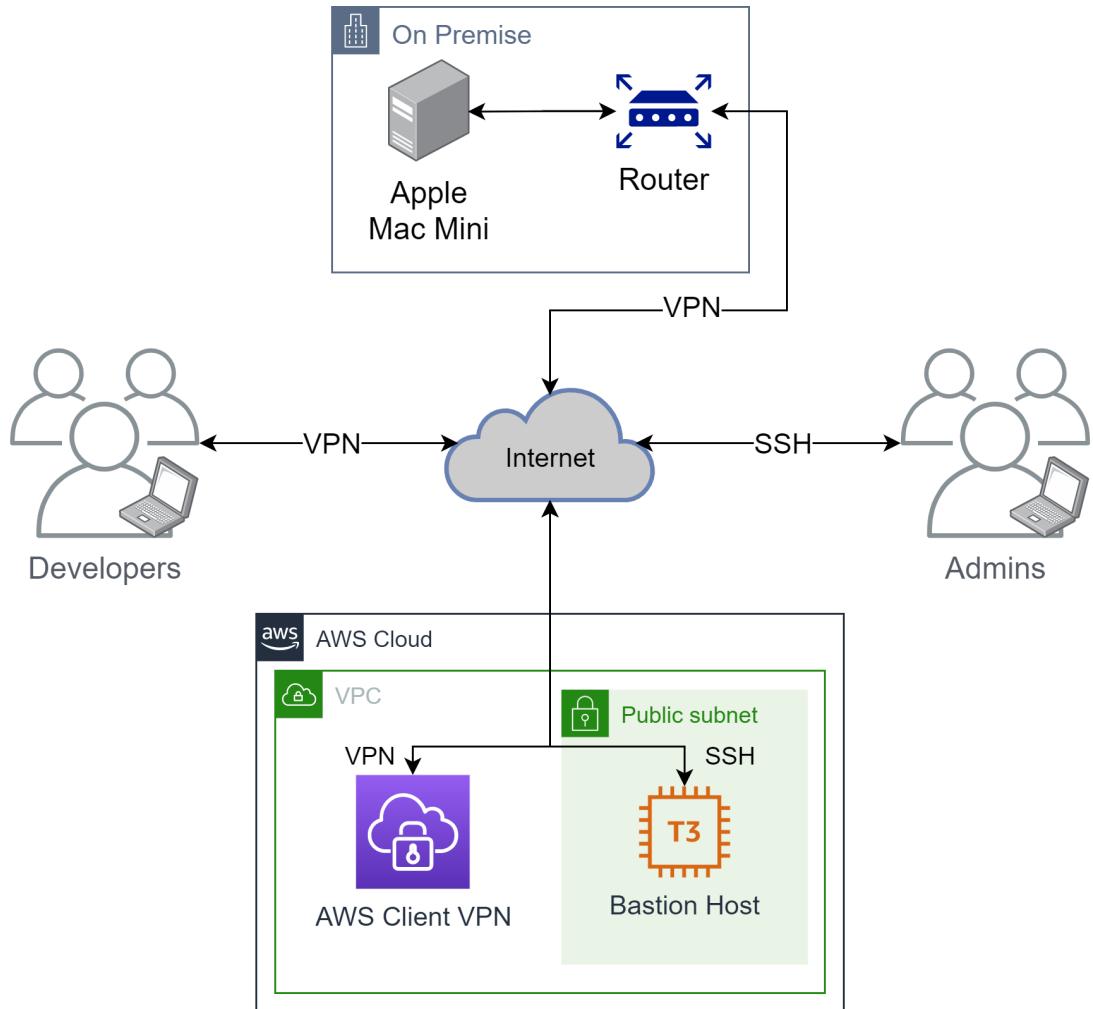


Figura 5.1: DevOps - Architettura Cloud - Esterna

L'architettura Cloud, descritta negli schemi 5.1 e 5.2, consiste nel raggruppamento delle risorse in una rete **AWS VPC** con le seguenti caratteristiche:

- **Region:** Europa Centrale (Francoforte)
- **CIDR Block:** 10.0.0.0/16
- **Availability Zone:** Multi-AZ
- **Subnets**
 - **Public:** 10.0.1.0/24
 - **Private (EC2):** 10.0.11.0/24 - 10.0.12.0/24 - 10.0.13.0/24
 - **Private (ELB):** 10.0.21.0/24 - 10.0.22.0/24 - 10.0.23.0/24
 - **Private (RDS):** 10.0.103.0/24 - 10.0.104.0/24 - 10.0.105.0/24
- **NAT Gateway:** singolo per traffico in uscita

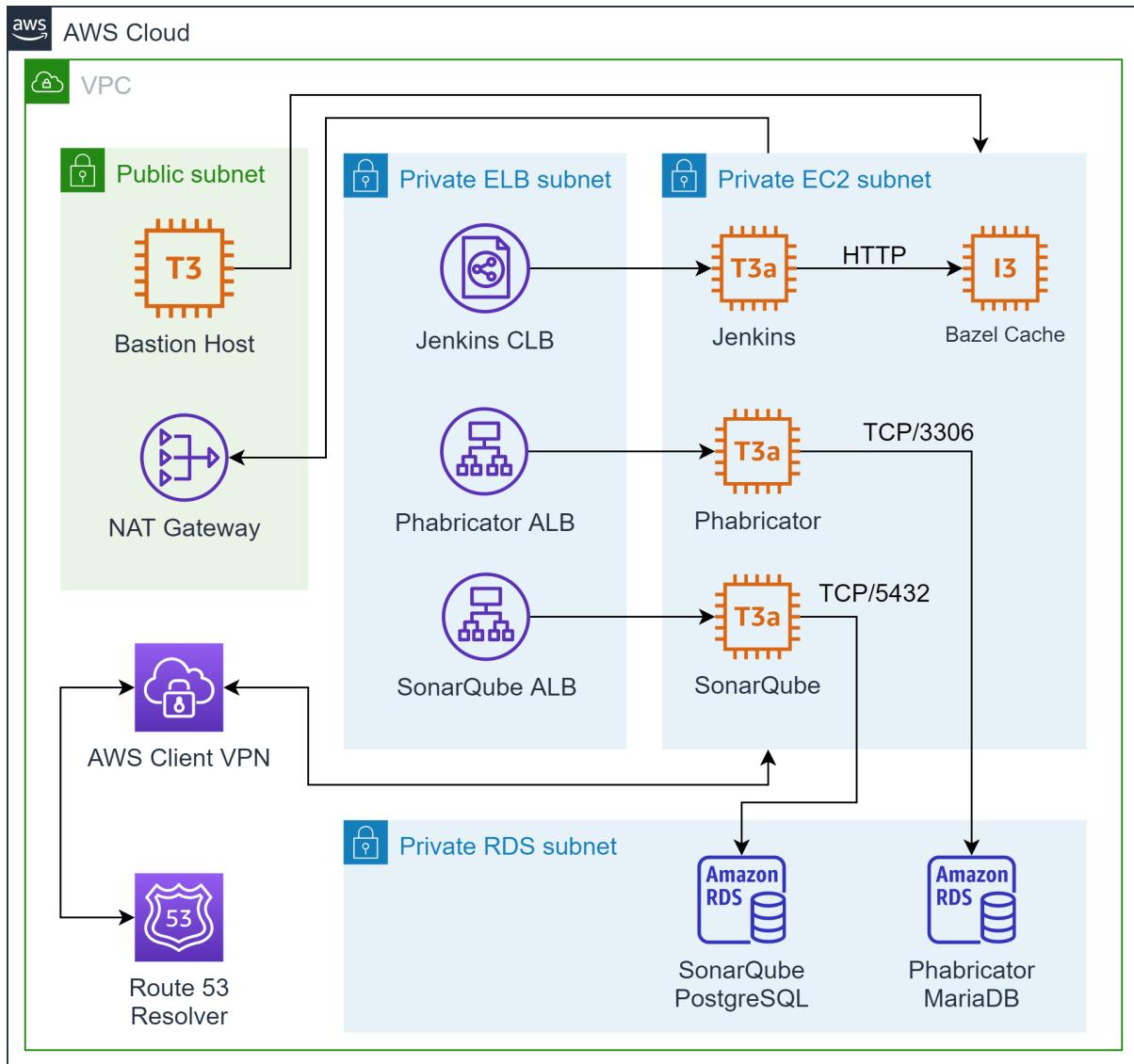


Figura 5.2: DevOps - Architettura Cloud - Interna

Accesso VPN All'interno della VPC è stata creata una istanza di **AWS Client VPN**, un servizio di Virtual Private Network basato su *OpenVPN* che permette, mediante il rilascio di appositi certificati firmati e l'associazione alle subnet della VPC, l'accesso alle reti private mediante un *tunneling* dedicato. Associando il servizio di VPN con il servizio di DNS privato **AWS Route 53 Resolver**, si ottiene una navigazione senza intoppi come se fossimo online. Il servizio è stato configurato per fornire IP della classe 10.1.0.0/16 e agganciarsi mediante una *route table* alle subnet private della VPC come descritte sopra.

Accesso SSH Un paradigma comune nella creazione di infrastrutture cloud è l'utilizzo di una macchina denominata **Bastion Host**, con IP pubblico (o DNS pubblico) raggiungibile, che funge da "controllore" sempre attivo per gli accessi SSH alle macchine private nella VPC. Utilizzare questo sistema permette di:

- Mantenere le macchine private sempre al sicuro in una subnet interna;

- Screamare gli accessi SSH mediante appositi controlli o esclusioni sul Bastion Host;
- Aprire all'esterno solo la porta SSH con un sistema a 2 step;
- Il resto del traffico dovrà passare solo dalla VPN precedentemente creata.

Nella infrastruttura definita, questa macchina viene creata mediante AWS EC2 con una istanza di tipo `t3.nano` (1 vCore, 1 GiB RAM) e storage EBS GP2 da 8 GiB, nella subnet pubblica (ottiene IP pubblico esterno). Associata a questa macchina si utilizza generalmente anche un DNS o Load Balancer con DNS per facilitare l'accesso, che non verrà descritto in questa architettura.

Macchine Virtuali Ogni strumento verrà installato in una singola istanza **AWS EC2** di grandezze e risorse differenti in base ai requisiti del singolo applicativo, specificatamente:

- **Bazel Cache:** Istanza `i3.large` (Storage Optimized), 2 vCore, 15.25 GiB RAM, 475GB SSD NVMe;
- **Phabricator/Jenkins/SonarQube:** Istanza `t3a.medium` (General Purpose), 2 vCore, 4 GiB RAM, Storage EBS GP2.

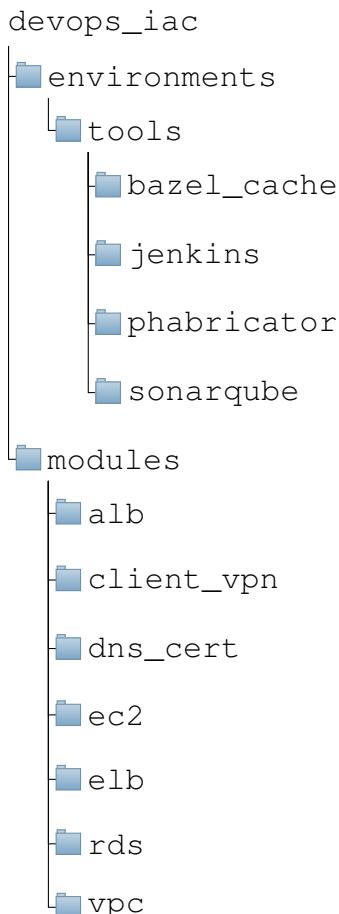
Database Gli strumenti che necessitano connessione ad un database relazionale (Phabricator e SonarQube), sono stati configurati utilizzando istanze *managed* di **AWS RDS** con il relativo engine di database necessario, in configurazione *Single AZ* e con backup periodici automatici. In particolare troviamo le seguenti configurazioni:

- **Phabricator MariaDB:** istanza `db.t3.small` (2 vCore, 2 GiB RAM), engine *MariaDB* 10.3.23, 20 GiB storage con autoscaling fino a 50 GiB;
- **SonarQube PostgreSQL:** istanza `db.t3.micro` (2 vCore, 1 GiB RAM), engine *PostgreSQL* 12.4, 20 GiB storage con autoscaling fino a 50 GiB.

5.3 Creazione Infrastruttura

5.3.1 Struttura *IaC*

L'infrastruttura è stata definita mediante l'uso di **Terraform** 0.14, provider AWS < 3.15 (a causa di un bug in versioni superiori riguardo l'utilizzo dell'ACL su Bucket S3, necessario all'uso del modulo di Bastion Host scelto). Il progetto ha la seguente struttura, creata per suddividere gli ambienti di deployment rispetto ai moduli riusabili:



Inizializzazione Terraform è stato inizializzato con la seguente configurazione, che permette di utilizzare un *Bucket S3* come storage remoto del suo stato, abilitando l'utilizzo da parte di più sviluppatori:

```
1 terraform {
2     required_version = ">= 0.14.0"
3
4     required_providers {
5         aws = "< 3.15"
6     }
7
8     backend "s3" {
9         bucket      = "perceptolab-devops"
10        region     = "eu-central-1"
11    }
12 }
```

Moduli Ogni modulo contenuto in `modules` segue una struttura di divisione delle risorse, dalle variabili in input e variabili in output, come da *Best Practice* su Terraform:

- **main.tf**: contiene tutte le risorse create dal modulo parametrizzate sulle variabili in ingresso e le fonti dati (*queried-only*);
- **variables.tf**: variabili in ingresso del modulo, permettono di riutilizzare il modulo generalizzando la sua implementazione;
- **output.tf**: variabili in uscita dal modulo, recuperate dalle varie risorse create dallo stesso e rielaborate per riutilizzo da parte di altre risorse o moduli richiamanti.

Prendendo come esempio il modulo di creazione di una generica macchina virtuale *EC2* possiamo trovare la seguente struttura nel suo file `main.tf`:

```

1 # Searches for the latest Amazon Linux 2 AMI
2 data "aws_ami" "amazon-linux-2" {
3     most_recent = true
4     (... )
5 }
6
7 resource "aws_instance" "ec2_instance" {
8     # Which Amazon Machine Image to use as base root device
9     ami = var.ami != "" ? var.ami : data.aws_ami.amazon-linux-2.
10      id
11
12     key_name      = var.key_name # SSH Key Name
13     instance_type = var.instance_type # EC2 Instance Type
14     (... )
15     dynamic "ephemeral_block_device" {
16         for_each = var.ephemeral_volumes # Configure EC2 Ephemeral
17           Volumes if available
18
19         content {
20             virtual_name = ephemeral_block_device.key
21             device_name  = ephemeral_block_device.value
22         }
23     }
24     (... )
25 }
```

Le variabili richiamate mediante la sintassi `var.<name>` sono definite nel file `variables.tf` in base al loro nome, tipo (stringa, numero, booleano, key-value, lista), descrizione e valore di default se applicabile:

```

1 variable "volume_size" {
2     type      = number
3     default   = 9
4     description = "EBS Root FS volume size in GiB"
5 }
6
```

```

7 variable "ephemeral_volumes" {
8   type      = map(string)
9   default    = {}
10  description = "AWS EC2 Instance Store ephemeral devices"
11 }

```

Mentre tutte le variabili di output risultanti dalla creazione delle risorse sopra definite vengono dichiarate all'interno del file `output.tf` seguendo la reference dell'oggetto creato dalla `resource` definita precedentemente:

```

1 output "id" {
2   value = aws_instance.ec2_instance.id
3 }
4
5 output "private_ip" {
6   value = aws_instance.ec2_instance.private_ip
7 }

```

Environment I moduli così creati con Terraform così inizializzato, permettono di essere composti in ciò che vengono definiti gli **ambienti**, ovvero set di moduli sia locali che remoti (mediante *Terraform Registry*) che definiscono un intero deployment. In questo progetto vi è un solo ambiente definito da diversi sotto-ambienti che definiscono a loro volta i singoli deployment per i singoli strumenti utilizzati, vi sarà quindi un `main.tf` generale dell'ambiente che definisce i sotto-ambienti da utilizzare:

```

1 # Define VPC module
2 module "vpc" {
3   source = "../../modules/vpc"
4   vpc_cidr_block = "10.0.0.0/16"
5 }
6
7 # Define Bastion Host via Terraform Registry
8 module "bastion-host" {
9   source  = "Guimove/bastion/aws"
10  (...)

11 }
12
13 # Defines SonarQube sub environment
14 module "sonarqube" {
15   source = "./sonarqube"
16   (...)

17 }

```

5.3.2 Struttura CaC

I vari strumenti sono stati configurati utilizzando **Ansible 2.9.x**, e l'integrazione da parte del plugin di **ansible.posix** in versione 1.1.1 e di **community.general** in versione 1.1.0, entrambi gestiti mediante *Ansible Galaxy*, un tool accessorio ad Ansible stesso che permette di gestire le dipendenze e i plugin all'interno dei ruoli e playbook definiti. Il progetto, contenuto

nello stesso repository, ha la seguente struttura:



Inventory Utilizzando una cloud platform, non possiamo sapere a priori l'indirizzo IP privato del server target né l'indirizzo IP esterno del Bastion Host, motivo per cui al posto di utilizzare un *inventory* statico, utilizziamo un **dynamic inventory**, formato da uno **script Python** `ec2.py` ed un file di configurazione `ec2.ini` forniti dai creatori di Ansible. Questo script sarà avviato da Ansible ed effettuerà delle query sulle risorse cloud, creando un *inventory* utilizzabile dal runtime del tool stesso.

Configurazione e Dipendenze Nella sottocartella `ansible`, oltre alla struttura descritta, vi sono dei file che definiscono la configurazione di Ansible stesso e le dipendenze dei nostri *playbook*. Tali file sono nominati `ansible.cfg` e `requirements.yml`, il primo definito con una singola modifica:

```
1 [defaults]
2 # Host key checking is enabled by default
3 host_key_checking = False
```

mentre il secondo presenta la seguente struttura, indicando le dipendenze relative ad *Ansible Galaxy*:

```
1 ---
2 collections:
3   - name: ansible.posix
4     version: 1.1.1
5   - name: community.general
6     version: 1.1.0
```

Playbooks Sempre nella sottocartella `ansible`, sono stati inseriti dei file YAML che definiscono i *playbooks* più esterni e quindi più specifici, al cui interno vengono utilizzati i *roles* definiti dalle cartelle create. Questi *playbooks* definiscono il nome dell'host target, le variabili in ingresso dall'esterno (al momento del richiamo di Ansible da terminale) e i ruoli da installare su

quel target host. Prendendo la configurazione di SonarQube come esempio, possiamo notare come il nome dell'host sia derivato dal **tag** della macchina **EC2** creata precedentemente:

```
1 ---
2 - name: SonarQube server setup
3   hosts: tag_Name_pl_tools_sonarqube_ec2
4   become: yes
5   gather_facts: no
6   vars:
7     ansible_ssh_common_args: -o ProxyCommand="ssh -W %h:%p -i
8       sshkey.pem ec2-user@{{ bastion }}"
9     postgres_host: "{{ postgres_host }}"
10    postgres_user: "{{ postgres_user }}"
11    postgres_pwd: "{{ postgres_pwd }}"
12  roles:
13    - base-docker
14    - sonarqube-service
```

Templating All'interno di ogni ruolo, ove necessario, sono presenti 2 sottocartelle, una contenente i *playbooks* che definiscono quel ruolo (*tasks*) e una seconda chiamata *templates* contenente eventuali file da copiare sulla macchina target che possono avere formato *j2*, ovvero che utilizzano il template engine integrato in Ansible chiamato **Jinja2**. Mediante questo engine è possibile sostituire all'interno dei file template i valori di determinati placeholder, ad esempio a partire dalle variabili in ingresso al *playbook* oppure a valori creati durante l'esecuzione (ad esempio password autogenerate).

5.4 Configurazione di Bazel Cache

L'utilizzo della cache remota in Bazel ci permette di velocizzare notevolmente le build effettuate consecutivamente, dove i cambiamenti incrementali sono piccoli in comparazione alla grandezza della codebase completa. La cache si presenta come un semplice **HTTP Server** con relativo storage, nel nostro caso il deployment è composto da:

1. **NGINX con WebDAV**: fornisce una interfaccia HTTP Plain con cui interagire con la cache su disco, mediante richieste HTTP *GET* e *PUT* gestite da Bazel stesso;
2. **Cache Cleaner**: *cron job* giornaliero che cancella i file più vecchi di 2 settimane (14 giorni) all'interno della cache su disco;
3. **Instance Storage**: la macchina EC2 scelta fornisce un disco SSD NVMe da 475 GiB utilizzato come storage "effimero", ovvero che viene eliminato ad ogni riavvio della macchina stessa, ma che offre performance estremamente elevate e consistenti. Sia NGINX che il Cleaner hanno accesso in R/W a tale storage.

NGINX L'istanza di NGINX viene installata mediante l'uso di un **container Docker**, immagine basata su `ubuntu:focal` mediante il quale viene installato il server e configurato per ascoltare su porta 80 e path `/cache`:

```

1 FROM ubuntu:focal
2 RUN apt-get update && \
3     DEBIAN_FRONTEND=noninteractive apt-get install -y --no-
        install-recommends nginx nginx-extras apache2-utils
4 # ...
5 CMD nginx -g "daemon off;"
```

Cleaner Anche il *Cache Cleaner* viene installato mediante creazione di una immagine e relativo **container Docker**, basato su alpine:3 dove viene semplicemente avviato il demone crond con il relativo cronjob configurato:

```

1 FROM alpine:3
2 # ...
3 COPY cronjobs /etc/crontabs/root
4 CMD ["crond", "-f", "-d", "8"]
```

Storage Per configurare ed utilizzare l'*Instance Storage* su una macchina EC2, si necessita di effettuare un *format* del volume, un *mount* su un determinato percorso (nel nostro caso /mnt/bazel_cache/cache) e di fornire i permessi necessari ai container creati mediante Docker:

```

1 (...)

2 - name: Create a ext4 filesystem on ephemeral storage
3   become: yes
4   community.general.filesystem:
5     dev: /dev/nvme0n1
6     fstype: ext4

7
8 - name: Mount ephemeral storage
9   become: yes
10  ansible.posix.mount:
11    src: /dev/nvme0n1
12    path: /mnt
13    fstype: ext4
14    opts: rw, noatime
15    state: mounted
16 (...)
```

La creazione delle 2 immagini citate (NGINX, Cleaner), l’istanziamento dei relativi container mediante Docker, e la configurazione dell’instance storage sono stati effettuati mediante l’utilizzo di un *role* dedicato con **Ansible**. A corredo della configurazione del server in se, viene creata anche una **entry nel DNS privato** che permette di accedere alla cache per le macchine interne alla VPC senza conoscerne l’IP privato.

5.5 Configurazione di Phabricator

Il deployment di Phabricator consiste in 2 componenti separate: il **Database** e l’istanza in se mantenuta sulla macchina virtuale EC2 creata. Usando Ansible definiamo un *playbook* composto da 2 parti:

1. **Environment**: effettua il setup del database gestendolo direttamente dalla macchina EC2 di Phabricator stesso, essendo l’istanza di RDS privata nella VPC. Crea l’utenza di Phabricator e da i permessi corretti per creare e gestire il proprio schema all’interno di MariaDB, effettua inoltre il setup dei file di configurazione di Phabricator localmente sulla macchina;
2. **Instance**: mediante l’utilizzo di Docker e l’immagine pubblica `phabricator/phabricator:stable`, crea un container a cui aggancia i file di configurazione ed un volume locale per mantenere i dati sulla macchina. Dopo la creazione, effettua il setup del database pilotando direttamente l’istanza creata di Phabricator usando una *shell* tramite Docker.

Database L’istanza di RDS basata su MariaDB viene configurata per creare una utenza con i permessi necessari ad effettuare le operazioni di cui necessita Phabricator. In particolare Ansible fornisce una azione chiamata `mysql_user` che permette di creare, modificare e cancellare utenze in un database *MySQL-based* (come MariaDB):

```

1 - name: Create Phabricator MariaDB user
2   become: yes
3   become_user: ec2-user
4   mysql_user:
5     name: "phabricator"
6     host: "%"
7     password: "{{ lookup('password', 'phabricator/.passwd'
8       length=32 chars=ascii_letters,digits,hexdigits') }}"
9     priv: "phabricator_%.*:<list of all permissions>"
```

Configurazione Phabricator necessita di un file `local.json` dove mantiene le configurazioni statiche, creato sostituendo i valori necessari dati in ingresso al *playbook* mediante il sistema di templating di Ansible, le cui configurazioni salienti sono le seguenti:

- `security.require-https`: forza l’utilizzo di HTTPS (*TLS Termination* su AWS ALB);
- `auth.email-domains`: abilita l’iscrizione solo da email dei domini permessi;
- `cluster.mailers`: configurazione del sistema di mailing interno per le notifiche;
- `mysql.<var>`: configurazione del database associato all’istanza;
- `recaptcha.<var>`: configurazione di *reCAPTCHA* per evitare accessi fraudolenti da parte di bot.

5.6 Configurazione di Jenkins

Il deployment di Jenkins si compone di una singola macchina dove l’istanza gestirà tutti dati necessari utilizzando lo storage integrato con AWS EBS, in cui mediante Ansible viene creato un container basato sull’immagine

`jenkinsci/blueocean:1.24.4` configurando l’ambiente nel seguente modo:

1. **Volumi**: per mantenere i dati del container sullo storage della macchine, viene creato un *Docker Volume* gestito e montato nel container sul path `/var/jenkins_home`;
2. **Container**: il container di Jenkins Blue Ocean viene creato esponendo le porte 80 e 50000, utilizzate rispettivamente per l’accesso alla Web UI e la connessione di *Agent* esterni mediante protocollo *JNLP*;
3. **Plugins**: a seguito della creazione del container vengono installati i plugins definiti in un file `plugins.txt`, che lista gli ID dei plugins come definiti nel repository di Jenkins;
4. **Arcanist**: per permettere a Jenkins di applicare le patch provenienti da Pull Request su Phabricator, viene installato il runtime PHP assieme alla libreria di *libphutil* e la CLI di *Arcanist* direttamente nel container.

Arcanist L’installazione di Arcanist all’interno del container avviene mediante l’utilizzo di `docker exec`, un comando di Docker che permette di avviare comandi utilizzando una shell interna ad un container. In questo caso verrà installato *PHP 7* dai repository di **Alpine Linux** (sistema su cui è basato l’immagine usata) e sia *libphutil* che *Arcanist* verranno installati clonando i rispettivi repository da Github:

```

1 - name: Install PHP for Arcanist
2   become: yes
3   become_user: ec2-user
4   command: "docker exec -it -u 0 jenkins apk add --no-cache --
5             update-cache php7 php7-curl php7-json"
6 (...)

6 - name: Clone Arcanist
7   become: yes
8   become_user: ec2-user
9   ignore_errors: yes
10  command: "docker exec -it jenkins git clone https://github.
           com/phacility/arcanist.git /var/jenkins_home/arcanist"
```

5.6.1 Agent su AWS EC2

Per gestire degli *Agent* utilizzando macchine virtuali AWS EC2, è necessario installare su Jenkins il rispettivo **plugin** che una volta configurato permette di gestire un **pool di server** in modo da mantenere le risorse disponibili quando necessario e di ridurre i costi se non ci si aspetta traffico sulla piattaforma. Il plugin si interfaccia con AWS mediante un **IAM Role** configurato ad-hoc per poter accedere alle API della piattaforma EC2 e creare così sia istanze *on demand* che di tipo *spot* per risparmiare sui costi dove possibile. Le macchine gestite da Jenkins vengono inoltre **distrutte quando non utilizzate**, permettendo un grado di cost saving consistente, specialmente in giorni non lavorativi.

La macchina che verrà creata dal plugin utilizzando come immagine di partenza un **AMI** (*Amazon Machine Image*) creato in precedenza, con al suo interno le seguenti dipendenze:

- OpenJDK 11
- Git con LFS

- Arcanist (con *PHP 7* e *libphutil*)
- Docker (per uso ambiente di build)
- Sonar Scanner (per analisi con *SonarQube*)

Jenkins, una volta creata la macchina mediante l'utilizzo del plugin EC2, accederà ad essa sfruttando **SSH** (nella stessa rete privata) e la comanderà come se fosse un semplice host linux remoto, permettendo di avviare comandi remoti, accedere ai suoi file e quindi implementare l'intera pipeline definita. Il **tipo di istanza** scelta per effettuare build dei servizi di backend è di tipo *c5a.2xlarge* con 8 vCore e 16 GiB di RAM.

5.6.2 Agent macOS On-Premise

Per effettuare compilazioni di artefatti, librerie e dipendenze per **Apple iOS** (OS mobile), è necessario utilizzare un ambiente basato su **macOS**, in questo caso un **Apple Mac Mini** gestito *on-premise* con installato l'ultimo aggiornamento di *macOS Big Sur* ed *XCode 12.4* che offre supporto ad *iOS 14.4*.

Sul Mac viene così configurato un **Jenkins Agent JNLP**, sfruttando la connessione VPN instaurata dal router *on-premise* verso la rete di AWS, non prima di aver effettuato il setup dell'ambiente installando le seguenti dipendenze:

- OpenJDK 8 (o 11)
- Git con LFS
- Arcanist (con *PHP 7* e *libphutil*)
- Bazel 3.7
- Eventuali altre dipendenze del sistema operativo (es. XCode Build Tools)

Da Jenkins verrà configurato l'agent, che genererà un file *agent.jar* ed una chiave di autenticazione univoca, entrambi da riportare localmente sulla macchina Mac. L'agent viene avviato mediante uno *script bash* installato come servizio di sistema all'avvio (mediante *launchctl*) con la seguente sintassi:

```

1 #!/bin/bash -l
2 /usr/bin/java -Xms256m -Xmx2048m -jar /Users/jenkins/Documents
   /scripts/agent.jar -jnlpUrl "https://jenkins.domain.com/
   computer/macos/slave-agent.jnlp" -secret "<secret>" -workDir
   "/Users/jenkins/build/"
```

5.7 Configurazione di SonarQube

Il deployment di SonarQube, come per Phabricator, si compone di una macchina virtuale EC2 e di un database relazionale. Mediante l'utilizzo di Ansible vengono effettuate le operazioni di configurazione dell'ambiente e di deployment del servizio in se, con il seguente schema operativo:

- **Database:** viene creato il database e l’utenza che userà SonarQube, a cui vengono dati i permessi di CONNECT/CREATE ed una password autogenerata in fase di creazione;
- **Environment:** nonostante Sonar utilizzi un DB relazionale esterno, internamente sfrutta una istanza di *ElasticSearch* che necessita di configurazioni aggiuntive nell’ambiente in cui viene creato. In particolare vengono modificate le seguenti variabili di sistema:
 - `vm.max_map_count`: numero massimo di aree di mapping di memoria che un processo può allocare;
 - `fs.file-max`: numero massimo di *file descriptors* aperti nel sistema;
 - `ulimit nproc`: numero massimo di processi aperti per singolo utente;
- **Deployment:** mediante l’utilizzo di Docker e l’immagine pubblica `sonarqube:lts`, viene creato un container collegato a 4 volumi dedicati a configurazioni, dati, estensioni e file di log. La configurazione del database creato viene passata come *variabili d’ambiente* nel container creato;
- **Plugins:** per supportare l’analisi di codice C++ nella versione *Community*, SonarQube necessita di 2 plugin esterni sotto forma di file `.jar`. Tali file vengono copiati all’interno del volume di estensioni creato precedentemente, così da permettere al server di rilevarli ed utilizzarli successivamente.

Database L’istanza RDS basata su engine PostgreSQL 12.4 viene configurata per creare una utenza che permetta connessione e creazione di schema, tabelle e quanto necessario a SonarQube stesso, in particolare l’utente `sonarqube` presenta la seguente azione nel *playbook*:

```

1 - name: Create SonarQube PostgreSQL user
2   become: yes
3   become_user: ec2-user
4   postgresql_user:
5     db: sonarqube
6     name: sonarqube
7     password: "{{ lookup('password', 'sonarqube/.passwd length
8       =32 chars=ascii_letters,digits,hexdigits') }}"
9     priv: "CONNECT/CREATE"
10    login_host: "{{ postgres_host }}"
11  (...)
```

Environment Per effettuare il setup delle variabili di sistema riguardo limiti di memoria, files e processi, viene sfruttato sia `sysctl` che `ulimit` in base alla variabile da modificare:

```

1 (...)

2 - name: Setup current session sysctl values - file-max
3   become: yes
4   command: sysctl -w fs.file-max=131072

5 - name: Setup current session ulimit values - file
6   become: yes
7   pam_limits:
```

```
9   domain: ec2-user
10  limit_type: '-'
11  limit_item: nofile
12  value: 131072
13  (...)
```

5.8 Creazione ambiente di build con Docker

Per effettuare compilazioni e renderle sempre riproducibili, indipendentemente dall'ambiente circostante come macchine locali degli sviluppatori o macchine remote per la *CI/CD*, viene creata una **Immagine Docker** contenente un ambiente controllato e sempre aggiornato sulle esigenze del progetto, che permetta di ottenere tutte le dipendenze in un singolo pacchetto e di evitare variabili inaspettate.

Tale immagine è basata su `debian:buster-slim` e durante la sua creazione vengono installate le seguenti dipendenze, utili alla compilazione e testing dei servizi di backend e mobile basati su linguaggi C++, Python e in parte Swift:

1. **C++**: LLVM con Clang e Clang-Tidy
2. **Python**: versione 3.7
3. **Bazel**: versione 3.7
4. **Swift**: versione 5.3, compilata per build su macchine *linux*
5. **JDK**: OpenJDK 11, per utilizzo con Bazel per *Code Coverage*

L'utilizzo locale di questa immagine viene semplificato mediante l'integrazione di uno **shell script** che automatizza le build mediante Bazel e permette ad Arcanist di effettuare l'analisi del *linting* ed avviare gli *unit tests* prima di creare una Pull Request su Phabricator. Nelle pipeline di CI/CD viene invece sfruttata l'immagine direttamente da Jenkins mediante il **Docker Plugin**, creando così un ambiente pronto all'uso ad ogni avvio del processo stesso, che viene eliminato alla sua conclusione.

Capitolo 6

La Pipeline di CI

In questo capitolo descriveremo lo schema, le tecnologie e la loro applicazione per la creazione della Pipeline di *Continuous Integration*. Sarà inoltre descritto il flusso completo e dettagliato del processo stesso, includendo le fasi pre/post dell'utilizzo della pipeline.

6.1 Tecnologie e Strumenti

6.1.1 Phabricator ed Arcanist: un flusso controllato

L'utilizzo di Phabricator assieme alla sua *CLI* Arcanist, permette agli sviluppatori e ai revisori di ottenere massima facilità di utilizzo del processo di sviluppo definito, astraendo tutte le fasi più tediose come la gestione del repository e fornendo una interfaccia grafica *user-friendly* ma al contempo efficace durante il suo utilizzo.

Di seguito elenchiamo le fasi che sfruttano questi 2 strumenti e come tali strumenti interagiscono col flusso generale:

1. Lo sviluppatore, partendo dalla *master branch*, apre localmente una *feature branch* e **committa** localmente tutte le modifiche volute;
2. A fine sviluppo, apre una **Pull Request** utilizzando il comando `arc diff`. Tale comando avvia di conseguenza `arc lint` e `arc unit` per effettuare linting e testing del codice modificato;
3. Una volta creata una PR in *draft* (bozza), Phabricator richiama il processo di CI, e ricevuto il risultato della build trasforma la PR in *Ready for Review*, notificando i revisori;
4. Al momento della accettazione della PR da parte del revisore utilizzando la Web UI di Phabricator, viene notificato lo sviluppatore che potrà effettuare il comando di `arc land`, unificando così le modifiche nella *master branch*;
5. Alla chiusura della PR, Phabricator richiama di nuovo la CI per effettuare il testing finale della codebase ed avviare l'analisi statica del codice.

Queste regole vengono configurate su vari livelli dello stack di sistema offerto da Phabricator, in particolare il sistema di tracciamento delle build automatiche viene gestito da **Harbormaster**, mentre le regole che definiscono i comportamenti in base a determinati eventi (es. avvio la build se arriva una nuova PR) vengono definite da **Herald**. La gestione della Code Review viene invece affidata a **Differential**, che agganciandosi al sistema di autenticazione ed utenze del core

di Phabricator, permette di controllare per singolo utente le Pull Requests aperte con annesse statistiche.

6.1.2 Jenkins: il motore del processo

La pipeline di *Continuous Integration* viene fatta muovere da Jenkins, il tool scelto per l'implementazione della cosiddetta *Continuous Build*, ovvero quella parte del processo di CI che si occupa di compilare, testare e fare reporting del codice modificato. All'interno di Jenkins viene creato un oggetto di tipo **Pipeline**, che permette, mediante la scrittura di un **Jenkinsfile**, di definire un processo sfruttando le risorse offerte dal sistema e dal linguaggio **Groovy**.

In particolare, la pipeline definita su Jenkins si compone delle seguenti fasi interne:

1. **Setup Environment**: notifica *BitBucket* della build in corso, recupera il token che verrà utilizzato da *Arcanist* per accedere alle Pull Request (diff);
2. **Checkout SCM**: Jenkins effettua il *checkout* del codice mediante **Git** su entrambi gli agent (macOS ed EC2) parallelamente;
3. **Apply Patch**: utilizzando *Arcanist*, Jenkins applica la patch relativa alla Pull Request (ricevuta da Phabricator) alla codebase principale. Questa azione viene fatta parallelamente su entrambi gli agent;
4. **Build**: su entrambi gli agent viene avviata la compilazione parallela, in particolare l'agent macOS si occupa dei *target* dipendenti dal suo ambiente, mentre l'agent EC2 del resto dei target (servizi di backend);
5. **Testing**: sull'agent EC2 viene avviata l'analisi degli *unit* ed *integration* tests, risultando in un file generato dalla pipeline stessa concatenando tutti i test avviati. Questo file, in formato **JUnit XML**, verrà poi analizzato dal plugin di Jenkins **Tests Analyzer** che fornisce l'andamento dei test nel corso delle run;
6. **Code Analysis**: questo step viene avviato solo al momento dell'unione con la codebase principale, sfrutta **Sonar Scanner** installato sull'agent EC2 ed analizza il codice dei servizi di backend come configurati nei file `.properties` definiti nel repository.

Patching

Jenkins sfrutta l'utilizzo di **Arcanist** per applicare la patch al codice principale relativa alla Pull Request in compilazione, come da richiesta pervenuta da Phabricator. Questa patch viene recuperata grazie ad un comando specifico e all'utilizzo del token di accesso ottenuto preventivamente grazie al tool integrato in Phabricator chiamato **Conduit**:

```
1 def applyArcanistDiff(conduitToken) {  
2     sh "bash -c 'git branch | grep -ve \"master\$\" | xargs git  
3         branch -D' || true"  
4     sh "arc --conduit-token ${conduitToken} patch --force --diff  
5         ${params.DIFF_ID}"  
6 }
```

Compilazione

Mediante l'utilizzo dell'immagine dell'ambiente di build, Jenkins crea un container all'interno del quale effettuare la build di tutti i target definiti da Bazel, utilizzando il *Docker Plugin* fornito da *CloudBees* che permette un controllo del *daemon Docker* remoto sulla macchina agent EC2:

```
1 stage ("Build Backend") {
2     agent { label "${EC2_AGENT_TYPE}" }
3
4     steps {
5         script {
6             docker.withRegistry("https://${IMAGE_REPO}") {
7                 docker.image("${BUILD_IMAGE}").inside("-v ${BAZEL_OUT}:${BAZEL_OUT}") {
8                     // Query bazel to get build targets
9                     BAZEL_TARGETS = sh (
10                         script: "bazel --output_user_root=${BAZEL_OUT} query //...",
11                         returnStdout: true
12                     ).trim().replace('\n', ' ')
13
14                     // Build backend services
15                     sh "bazel --output_user_root=${BAZEL_OUT} build --remote_cache=${BAZEL_CACHE_URL} ${BAZEL_TARGETS}"
16                 }
17             }
18         }
19 }
```

Testing

Dopo aver avviato tutti i test sempre grazie all'utilizzo di Bazel, Jenkins "raccoglie" tutti i report generati dai singoli gruppi di test, che vengono a loro volta parsati e aggregati in un singolo file che comprenderà quindi tutti i test suddivisi per sorgente, con relativi casi di successo o fallimento. Questo tipo di logica è scritta in **Groovy** con l'utilizzo delle librerie base di Jenkins e del linguaggio stesso:

```
1 def createTestOutputXml(outputFile) {
2     def testFiles = findFiles(glob: 'bazel-testlogs/**/test.xml')
3     def testSuites = testFiles.collect {
4         parseAndReplaceTestXmlFile(it.path) }
5         // ...
6
7     def parentNode = new Node(null, "testsuites")
8     // ...
9     testSuites.each {
10         sumValueInMap(parentNode.attributes(), "total", it.
11             total)
12 }
```

```

10    // ...
11    it.suites.each { node -> parentNode.append(node) }
12  }
13  nodePrinter.print(parentNode)
14
15  writeFile(file: outputFile, text: stringWriter.toString())
16 }
```

6.1.3 Docker: ambiente di testing unificato

L'immagine Docker descritta nel capitolo precedente, contenente tutte le dipendenze necessarie ad effettuare una compilazione con successo di tutti i servizi di backend nel progetto, viene sfruttata sia localmente grazie all'integrazione mediante Arcanist (per *linting* e *unit testing*) sia remotamente mediante Jenkins. Seguendo infatti il codice della sezione precedente alla attuale, notiamo che lo strumento sfrutta l'immagine grazie al **Docker Plugin**, creando un container durante la compilazione degli artefatti e distruggendolo alla conclusione.

6.1.4 SonarQube: controllo qualità

Dopo la fase di *merge* della Pull Request all'interno della *master branch*, Phabricator avvia nuovamente la pipeline di *Continuous Build* su Jenkins ma con parametri differenti. Jenkins infatti non applicherà più le patch al codice, ma effettuerà l'analisi statica e dinamica del codice sfruttando **Sonar Scanner** e l'istanza di **SonarQube** installata. L'analisi viene effettuata automaticamente solo sui servizi di backend in **Python** e **C++**, a causa di una limitazione nell'uso commerciale della analisi di codice Swift, e per ogni servizio viene definito un file di configurazione `.properties` come segue:

```

1 sonar.projectKey=service-name
2 sonar.projectName=Sphere - Service Name
3
4 sonar.sources=service/
5 sonar.tests=service/
6 sonar.test.inclusions=**/*-test.cc
7 sonar.exclusions=**/*.proto,**/BUILD,**/*-test.cc
8 sonar.sourceEncoding=UTF-8
9
10 sonar.cxx.clangtidy.reportPath=bazel-out/k8-fastbuild/bin/
     service/**/*.clang-tidy.report
```

Punto saliente della configurazione è sicuramente la dicitura relativa al plugin installato in precedenza per il supporto al codice C++ (CXX Plugin), che permette di integrare le analisi effettuate mediante lo *static analyzer* **Clang Tidy**.

Integrazione nella Pipeline

Per avviare le analisi con Sonar Scanner, Jenkins integra un plugin dedicato che permette di definire a livello di sistema la configurazione desiderata e il server target (l'istanza di SonarQube). Oltre a dover definire l'environment, il codice scorre tutti i file `.properties` che trova all'interno di una sottocartella dedicata, così da analizzare automaticamente tutti i servizi ad ogni *merge*:

```
1 withSonarQubeEnv(installationName: "${env.  
2   SONARQUBE_SERVER_NAME}", envOnly: true) {  
3   script {  
4     def configs = findFiles(glob: 'continuous-integration/  
5       sonarqube/*.properties')  
6     configs.each {  
7       sh "sonar-scanner -Dsonar.host.url=${env.  
8         SONAR_HOST_URL} -Dsonar.login=${env.  
          SONAR_AUTH_TOKEN} -Dproject.settings=${it}"  
    }  
  }  
}
```

6.2 Diagramma della Pipeline

6.2.1 Flusso di Sviluppo

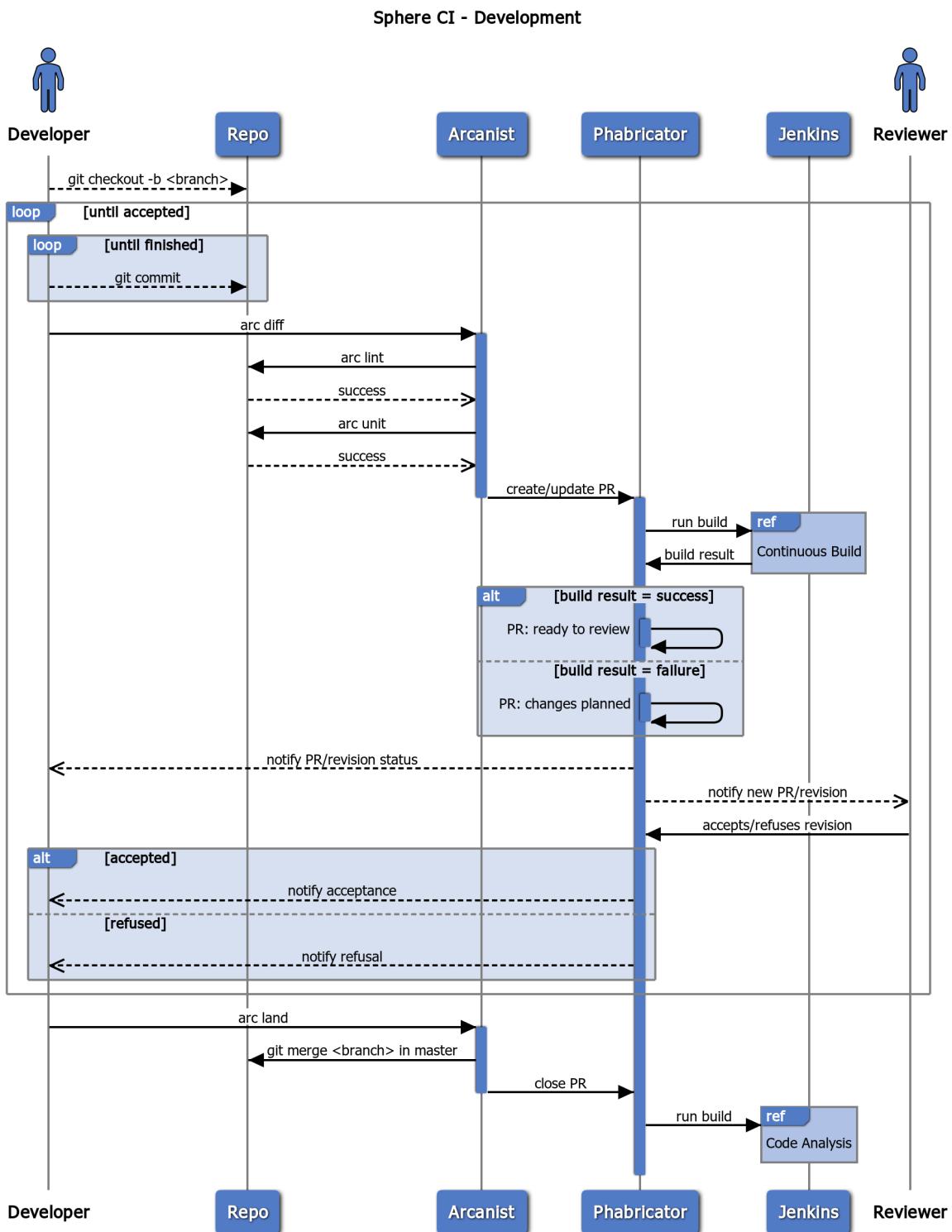


Figura 6.1: Sphere CI - Development Flow

6.2.2 Continuous Build

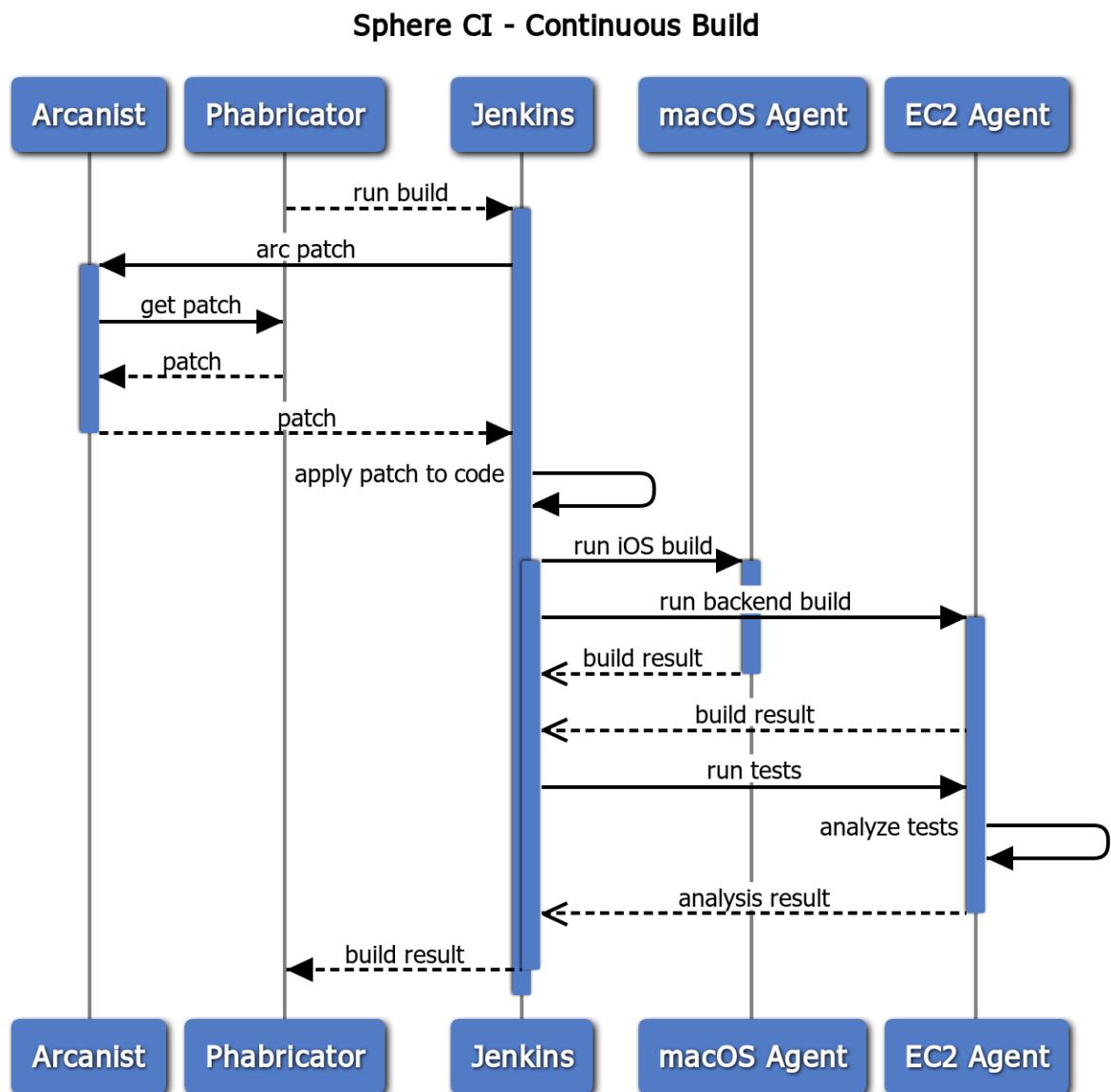


Figura 6.2: Sphere CI - Continuous Build

6.2.3 Analisi del Codice

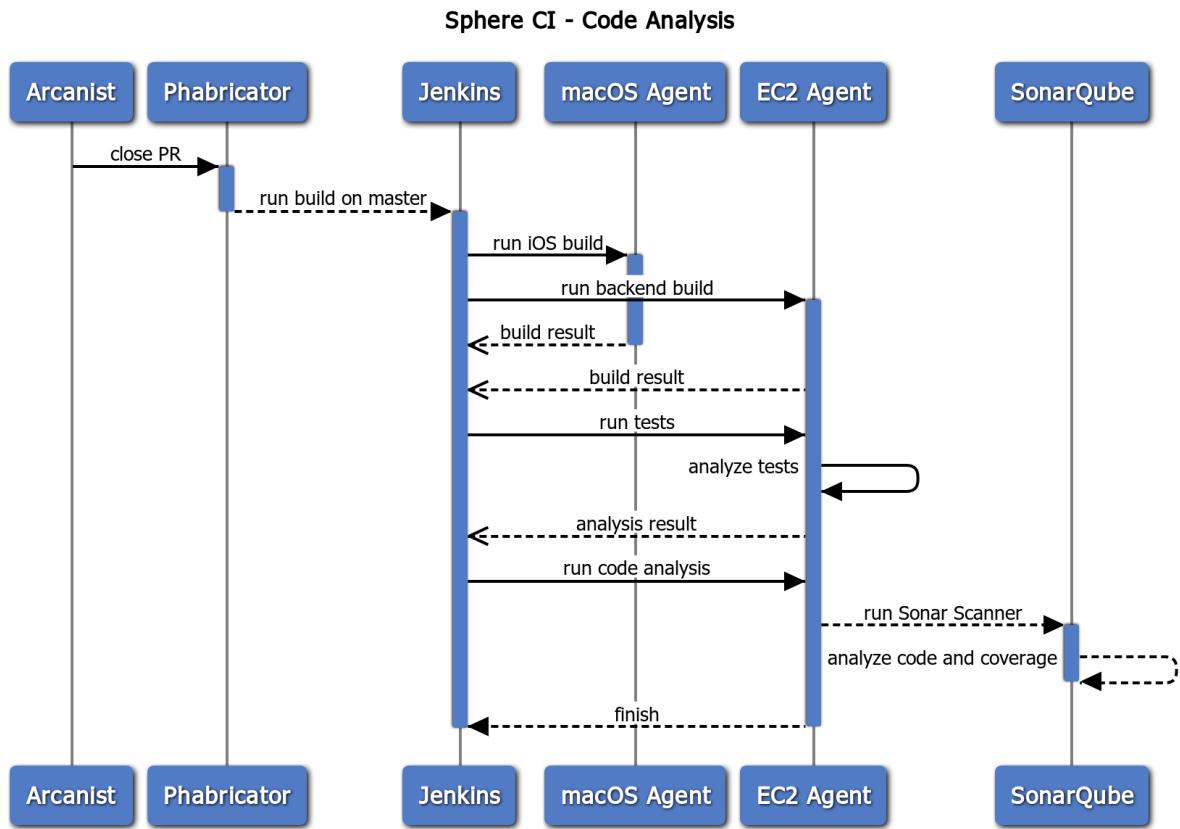


Figura 6.3: Sphere CI - Code Analysis

6.3 Testing e Analisi del Codice

6.3.1 Unit e Integration Tests

L'esecuzione degli *unit* ed *integration tests* permette di rilevare problemi precocemente durante le prime fasi di revisione di una Pull Request. Tali tests vengono definiti in fase di sviluppo, integrati nel build system mediante definizione di regole relative a *Bazel*, e poi eseguiti dal processo di *Continuous Build* da Jenkins.

Il **Tests Analyzer Plugin**, illustrato in figura 6.4, permette di aggregare, visualizzare ed analizzare l'andamento dei test in base alle build effettuate (le ultime 10 in questo caso), utilizzando una semplice interfaccia integrata nella schermata di pipeline su Jenkins. Per sfruttare questo plugin il formato di output dei test deve necessariamente essere **JUnit XML**, inoltre il plugin permette di visualizzare anche il **tempo di esecuzione** dei singoli test, individuando quelli che necessitano di ottimizzazione o *refactoring* perché troppo lunghi.

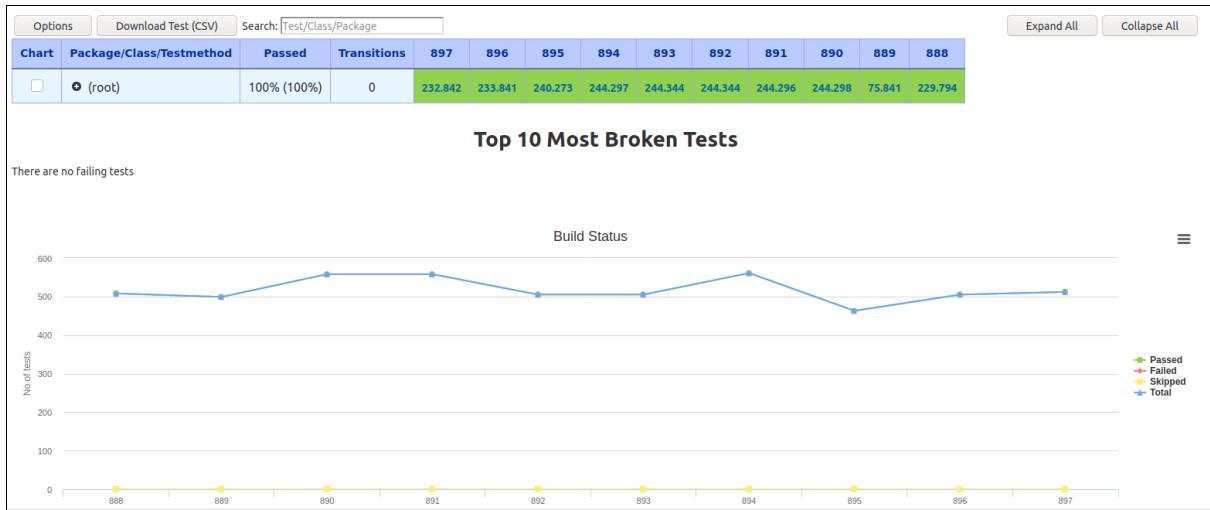


Figura 6.4: Jenkins - Tests Analyzer Plugin

6.3.2 Code Coverage

Una analisi decisamente importante in fase di testing del codice è la *Code Coverage*, ovvero quante righe di codice, *branches* e rami sono stati coperti dai test definiti. Più è alta la coverage, più è possibile **scovare potenziali bug** avendo "scoperto" tutti i rami possibili del codice, ma **non indica la correttezza dei test** in se che potrebbero mancare di alcuni casi limite non gestiti neanche dal codice stesso.

Purtroppo, a causa di problemi durante l'implementazione della analisi di coverage sfruttando il build system scelto (Bazel) e a causa di bug e problemi interni allo stesso (file incompleti, necessità di dipendenze), **non** si è stati in grado di integrare con successo una analisi di *Code Coverage* funzionante al momento della stesura di questo documento.

6.3.3 Quality Gates

Per definire un livello di "qualità" del codice in SonarQube, vengono utilizzati i cosiddetti *Quality Gates*, ovvero soglie e definizioni create a priori che permettono poi a Sonar di definire se il progetto ha passato o meno quelle soglie, dando una idea generale della qualità del progetto stesso.

Prendendo come base il Quality Gate di default di SonarQube (chiamato *Sonar way*), lo si è adattato semplicemente per integrare la mancanza di analisi della *Code Coverage*, quindi eliminando la necessità che quel valore sia superiore all'80%, come visibile in figura 6.5. Il Quality Gate di default fornisce già da se una indicazione valida di qualità, dando come soglia minima un voto pari ad **A** per ogni parte della analisi di SonarQube, sotto il quale il controllo fallisce e bisogna reagire di conseguenza.

Sphere QG

Rinomina Copia Imposta come default Elimina

Condizioni Aggiungi una condizione

Solo le misure del progetto sono controllate con le soglie. Le misure per moduli, packages e classi non sono comprese.

Metric	Operator	Error
Copertura sul nuovo (UT)	e' minore di	0,0%
Duplicated Lines on New Code	e' maggiore di	3,0%
Maintainability Rating on New Code	is worse than	A
Reliability Rating on New Code	is worse than	A
Security Rating on New Code	is worse than	A

Figura 6.5: SonarQube - Quality Gate

6.3.4 Risposta a cambiamenti nella qualità

A seguito di una analisi effettuata su SonarQube, il sistema potrebbe individuare potenziali bug, code smells, duplicazioni o indicazioni che potrebbero interessare o meno in base alla loro gravità.

Add a nested comment explaining why this function is empty or complete the implementation. See Rule 3 mesi fa L69 suspicious

Code Smell Critiche Aperito s.renzo@perceptolab.com 5min effort Commento

Either remove or fill this block of code. See Rule 3 mesi fa L99 suspicious

Code Smell Maggiori Aperito s.renzo@perceptolab.com 5min effort Commento

Complete the task associated to this TODO comment. See Rule 3 mesi fa L119 cwe

Code Smell Informativo Aperito s.renzo@perceptolab.com Commento

Complete the task associated to this TODO comment. See Rule 3 mesi fa L122 cwe

Code Smell Informativo Aperito s.renzo@perceptolab.com Commento

4 of 4 shown

Figura 6.6: SonarQube - Issues

Tali **issues** individuate da Sonar verranno **automaticamente assegnate** da risolvere **allo sviluppatore che le ha introdotte**, come definito dal sistema di versioning del codice (Git), mentre non verranno assegnate se l'analisi è stata effettuata per la prima volta e quindi non ha uno stato precedente con cui confrontare. Rimane compito dello sviluppatore controllare ad ogni *merge* il risultato ottenuto dalla analisi di SonarQube, o in generale controllare le *issues* a lui assegnate, da risolvere alla iterazione successiva o prima di sviluppi ulteriori che possano intaccare il codice analizzato. Un esempio di issues è visibile in figura 6.6 dove sono presenti 4 issues pre-assegnate a chi le ha introdotte nel codice.

6.4 Risultati

L'analisi dei test effettuati nel codice e l'analisi statica del codice ha permesso di ottenere una **migliore visibilità** di ciò che potrebbe impattare il progetto nel medio e lungo termine. Grazie alla integrazione di tutta la pipeline di *CI* ora gli sviluppatori riescono a **risolvere preventivamente i problemi** che sorgono durante la compilazione, quando prima spesso si rischiava di unificare del codice che in realtà non funzionava, per poi accorgersene in fase troppo tardiva e risultava necessario agire per risolvere i problemi introdotti.

La facilità di utilizzo di tali analisi, mediante l'introduzione di uno strumento come Phabricator ed Arcanist, assieme ad uno strumento utilizzato in tutto il mondo come Jenkins, ha permesso quindi di **aumentare esponenzialmente la qualità del codice** prodotto, renderlo sempre più visibile a tutto il team che risulta così coinvolto attivamente nelle fasi di *Code Review*, *Continuous Build* e *Code Analysis*.

Un guadagno importante è stato inoltre la **riduzione notevole nelle tempistiche di compilazione**, con guadagni nell'ordine del 300% nei casi medi e fino al 500% nei casi migliori, rispetto al processo legacy illustrato. Con il nuovo processo infatti non troviamo più colli di bottiglia dovuti alla compilazione dei servizi di backend, pur avendo più step e più operazioni nella pipeline, con lo step più lento (la build su macOS) che impiega circa 1.30 minuti nel caso migliore. L'unica casistica in cui si possono riscontrare rallentamenti è nella prima build della giornata, dovuti al *cold start* delle istanze *agent* create da Jenkins.

Riprendendo infine gli obiettivi iniziali prefissi, ed applicandoli al processo di *Continuous Integration*, possiamo stilare una tabella di conseguimento che denota il loro raggiungimento completo (con tutte le variabili del caso):

Obiettivo	Raggiungimento
Analisi dei Requisiti per i processi CI	✓
Pipeline di CI per i servizi di Backend e Mobile	✓
QA basato sulla analisi dei test e del codice	✓
Infrastruttura basata su AWS per gestire il processo	✓

Tabella 6.1: Sphere - Raggiungimento Obiettivi CI

6.5 Esperienza Applicativa

Durante il corso del design ed implementazione della pipeline di *Continuous Integration* vi sono state diverse modifiche nell'approccio ad uno dei problemi fondamentali, ovvero i tempi di compilazione.

Partendo come spunto dal processo legacy precedente, si era provato inizialmente a **riutilizzare** il servizio di **AWS CodeBuild** con poco successo, anche sfruttando i sistemi di caching remoto o tramite file system, a causa della astrazione che fa il sistema non poteva garantire l'utilizzo della stessa macchina e quindi dello stesso ambiente. Questo fatto causava problemi a Bazel che spesso e volentieri ricompilava intere dipendenze per un cambio solo dell'ambiente di build, allungando le tempistiche al pari del processo precedente.

Un altro problema sorto durante lo sviluppo è stata la **connessione VPN** utilizzata sull'**Apple Mac Mini**, inizialmente gestita a livello di client nel sistema operativo, che frequentemente si disconnetteva e faceva fallire delle build avviate da Jenkins specialmente nelle fasi iniziali (ma dopo lunghi timeout). A seguito di un ammodernamento dell'infrastruttura di rete *on premise*, si è poi spostata la connessione VPN a livello di router, eliminando così questa variabile e risolvendo definitivamente il problema.

Il resto del processo è stato visto essere molto solido e ben riutilizzabile indipendentemente dal progetto, o con minimi cambiamenti dovuti più all'ambiente di compilazione che al processo in se. Punto interessante è di sicuro l'**integrazione locale con Arcanist**, l'unico vero punto dipendente dal progetto poiché **sfrutta Bazel** per effettuare le sue analisi, rendendo più difficoltosa l'integrazione con altri progetti (ma già effettuata con build system differenti con successo e poco effort).

Capitolo 7

La Pipeline di CD

In questo capitolo descriveremo lo schema, le tecnologie e la loro applicazione per la creazione della Pipeline di *Continuous Delivery*. Sarà inoltre descritto il flusso completo e dettagliato del processo stesso.

7.1 Tecnologie e Strumenti

7.1.1 Git e Bitbucket: il *trigger*

Il processo di *Continuous Delivery* non viene avviato in modo automatico, la decisione spetta al team di sviluppo che, volendo effettuare una release di un determinato servizio o set di servizi, **tagga** il repository sulla *master branch*. Il tag viene creato mediante *Git* e segue il formato `<service>/<version>`, e viene poi *pushato* su *BitBucket*.

A seguito di un nuovo *tag*, BitBucket procede automaticamente a mandare un **webhook** verso *Jenkins*, ovvero una richiesta contenente le informazioni salienti riguardo il tag appena creato e che verrà interpretata dal **BitBucket Plugin** installato.

7.1.2 Jenkins: il motore del processo

La pipeline di *Continuous Delivery* viene fatta gestire interamente da Jenkins, che si occupa di ricevere il *trigger* per il suo avvio, i relativi dettagli alla compilazione da effettuare e a procedere con la compilazione stessa e l'invio degli artefatti al repository scelto. All'interno di Jenkins viene creato un oggetto di tipo **Pipeline**, che permette, mediante la scrittura di un **Jenkinsfile**, di definire un processo sfruttando le risorse offerte dal sistema e dal linguaggio **Groovy**.

In particolare, la pipeline definita su Jenkins si compone delle seguenti fasi interne:

1. **Webhook Parsing**: una volta ricevuto il *webhook* da parte di BitBucket, grazie al plugin dedicato, la pipeline lo parsa ed estrae il nome del servizio, la versione voluta, i target necessari alla sua compilazione e l'immagine Docker di destinazione;
2. **Remote Build**: mediante l'utilizzo dell'**AWS CodeBuild Plugin**, Jenkins fa *offloading* della compilazione sfruttando il servizio managed di AWS ed inviando i dettagli necessari al suo corretto funzionamento (dallo step precedente);

3. **Notification:** alla fine del processo di compilazione su AWS CodeBuild, mediante il **Google Chat Plugin**, Jenkins notifica i team di sviluppo della avvenuta *delivery* dell’artefatto richiesto oppure del fallimento del processo.

Parsing

Per effettuare il *parsing* del **webhook** ricevuto da BitBucket il sistema funziona su due livelli: un **job** che sfrutta il BitBucket Plugin e che quindi può essere avviato dallo stesso, e codice *Groovy* nella pipeline di delivery che effettua il parsing vero e proprio. Questa separazione è stata necessaria a causa di un bug presente nel plugin, che non permette di avviare direttamente un oggetto di tipo *pipeline* su Jenkins, quindi viene avviato un *job* che a sua volta avvierà la pipeline di CD.

In particolare, il codice nella pipeline che effettua il parsing del *tag* in arrivo, sfrutta le librerie standard del linguaggio (derivate dalla JVM), ed una *map* contenente i servizi abilitati alla delivery assieme ai loro dettagli:

```

1 def (r, t, serviceName, serviceVersion) = params.SCM_REF.split
2 ('/')
3
4 if (servicesMap.containsKey(serviceName)) {
5     buildData = servicesMap[serviceName]
6     buildData["version"] = serviceVersion
7     buildData["scmRef"] = "refs/tags/${serviceName}/${serviceVersion}"
8 } else {
9     currentBuild.result = 'ABORTED'
10    error('The received tag contains an unsupported service')
11 }
```

Remote Build

Per permettere a Jenkins di effettuare compilazioni remote sfruttando AWS CodeBuild si è dovuto installare il relativo plugin ufficiale fornito nel repository, assieme alla configurazione di una utenza mediante **AWS IAM** con i relativi permessi per accedere al servizio di CodeBuild. Grazie al plugin è inoltre stato possibile inserire delle variabili d’ambiente prima dell’avvio del processo di compilazione, permettendo quindi di parametrizzarlo e renderlo generico:

```

1 awsCodeBuild (
2     credentialsType: 'jenkins',
3     credentialsId: "${AWS_CREDENTIALS_ID}",
4     projectName: "${CODEBUILD_SPHERE_CD_PROJECT_NAME}",
5     region: "${CODEBUILD_SPHERE_CD_REGION}",
6     sourceVersion: "${buildData.scmRef}",
7     envVariables: "[${SERVICE}, ${buildData.folder}], {VERSION,
8         ${buildData.version}}, {TARGET, ${buildData.target}}, {
9             IMAGE_NAME, ${buildData.image}}]"
```

7.1.3 AWS CodeBuild: compilazioni remote

Il servizio di *AWS CodeBuild* permette di creare delle pipeline per la compilazione di codice di vario genere, a partire da una definizione in formato **YAML** ed un ambiente creato mediante una immagine Docker. Per effettuare la *delivery* è stato scelto l'utilizzo di CodeBuild poichè non vi è necessità di tempi di compilazione molto stretti, né di caching durante la compilazione, quindi ogni avvio è come un *fresh start* e permette di ottenere compilazioni sempre ottimizzate in tempi discreti.

Se per l'ambiente di compilazione viene sfruttata la stessa immagine Docker utilizzata nel processo di *Continuous Integration*, la specifica di compilazione si definisce con un file `buildspec.yml` nel repository che viene letto da AWS CodeBuild in fase di avvio di una build. Tale file definisce i vari step da effettuare, tra cui l'inizializzazione di Docker all'interno del container, la compilazione degli artefatti mediante *Bazel*, e il caricamento degli artefatti (immagine in `.tar`) in una immagine Docker finale che verrà poi pushata sul registro scelto:

```
1 version: 0.2
2
3 phases:
4   pre_build:
5     commands:
6       - nohup /usr/sbin/dockerd --host=unix:///var/run/docker.sock --host=tcp://127.0.0.1:2375 --storage-driver=overlay2 &
7       - timeout 15 sh -c "until docker info; do echo .; sleep 1; done"
8   build:
9     commands:
10      - bazel build -c opt //:$SERVICE:$TARGET.tar
11      - docker load -i bazel-bin/$SERVICE/$TARGET.tar
12   post_build:
13     commands:
14       - docker tag bazel/$SERVICE:$TARGET $SPHERE_REPO/$IMAGE_NAME:$VERSION
15       - aws ecr get-login-password --region $SPHERE_REGION | docker login --username AWS --password-stdin $SPHERE_REPO
16       - docker push $SPHERE_REPO/$IMAGE_NAME:$VERSION
```

7.1.4 Docker: ambiente e artefatti

Nella pipeline di *Continuous Delivery*, Docker viene sfruttato in due modalità differenti: per la creazione dell'ambiente di compilazione, sfruttando la stessa immagine usata nella *Continuous Integration*, sia come artefatti in output alla pipeline stessa, ovvero il prodotto che andrà poi rilasciato.

Ottenerne immagini Docker come artefatti finali permette di sfruttare i sistemi di deployment *cloud native*, come *Kubernetes*, ed evitare ulteriori problemi di dipendenze essendo totalmente *self contained*. Tali artefatti vengono inoltre inviati ad un **Docker Registry** in cloud, ovvero il

servizio di **AWS Elastic Container Registry** (ECR), che permette di ottenere alta disponibilità online ed un sistema integrato di **Vulnerability Analysis** per ogni immagine *pushata*.

7.2 Diagramma della Pipeline

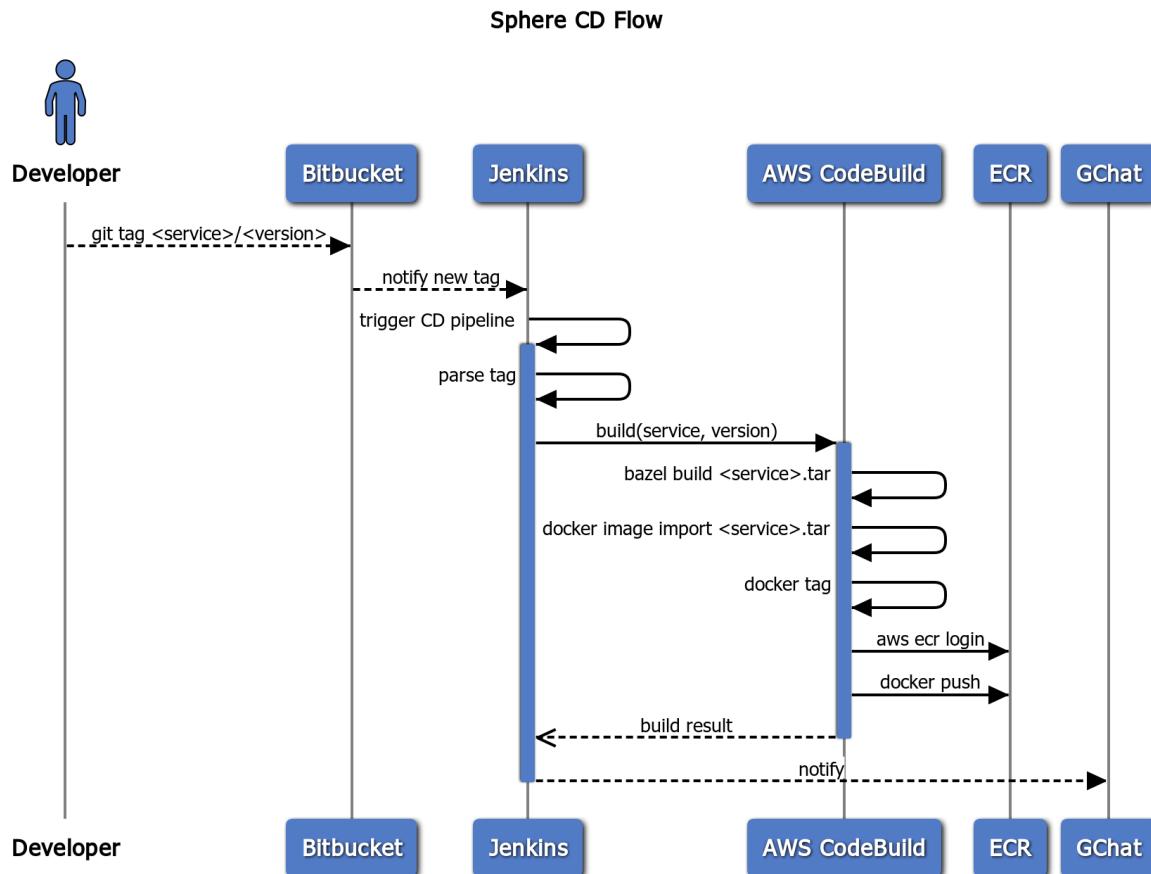


Figura 7.1: Sphere - Continuous Delivery

7.3 Analisi delle Vulnerabilità

Come accennato nella sezione relativa alla delivery degli artefatti, le immagini dei servizi compilati vengono gestite da **AWS Elastic Container Registry** (ECR), un Docker Registry remoto altamente disponibile che integra una feature molto utile al fine di analisi della sicurezza dei sistemi.

Ad ogni *push* di una nuova immagine il registry effettua una **Analisi Statica delle Vulnerabilità**, seguendo come indicazione la lista delle **Common Vulnerabilities and Exposures**[32] (CVE), ed utilizzando come strumento di analisi **Quay Clair**[33], un software *open source* che permette di analizzare qualsiasi container che segua la specifica **OCI**. Nel caso in cui la gravità della vulnerabilità non sia contenuta nel repository ufficiale dei CVE, ECR utilizza il **Common Vulnerability Scoring System**[34] (CVSS), mediante il quale computa un punteggio

e lo confronta con il **National Vulnerability Database**[35] (NVD) del NIST.

Per configurare l’analisi statica delle vulnerabilità di un repository in ECR, basta abilitare tale funzione al momento della creazione dello stesso o in un secondo momento, oppure è possibile anche effettuare analisi *on demand* manuali in un secondo momento.

7.4 Risultati

L’introduzione di una pipeline di *Continuous Delivery* ha permesso di **rimuovere** un fattore molto rilevante a livello di rilascio, ovvero la **dipendenza dal team stesso** per la compilazione e la delivery degli artefatti necessari al deployment. Precedentemente infatti il processo veniva effettuato manualmente e sulle macchine locali degli sviluppatori, che potevano impiegare diverse decine di minuti a compilare tutte le dipendenze, compilare gli artefatti software e deliverarli manualmente sul Docker Registry. Questo processo non era inoltre descritto ne documentato, rendendo ancora più difficile la delivery in casi di urgenza, dove si rendeva necessario chiedere personalmente a chi se ne fosse occupato fino a quel momento.

In aggiunta alla velocità e standardizzazione del processo, l’introduzione delle analisi sulle vulnerabilità ha permesso di migliorare la qualità finale degli artefatti rilasciati, rilevando ed aggiornando così eventuali dipendenze che potrebbero causare problemi di sicurezza in fase di rilascio ed utilizzo del sistema.

Riprendendo infine gli obiettivi iniziali prefissi, ed applicandoli al processo di *Continuous Delivery*, possiamo stilare una tabella di conseguimento che denota il loro raggiungimento completo (con tutte le variabili del caso):

Obiettivo	Raggiungimento
Analisi dei Requisiti per i processi CD	✓
Pipeline di CD per i servizi di Backend	✓
Infrastruttura basata su AWS per gestire il processo	✓

Tabella 7.1: Sphere - Raggiungimento Obiettivi CD

7.5 Esperienza Applicativa

Le scelte effettuate in materia di *Continuous Delivery* sono state condizionate dal volere di mantenere semplice il processo, come ad esempio l’utilizzo di un servizio *managed* per la compilazione come **AWS CodeBuild** al posto dello stesso sistema utilizzato nella pipeline di *Continuous Integration*. Questo ha permesso di poter sviluppare un processo high level molto generico e semplice, e di poter poi modificare la fase di compilazione agendo solo sulla specifica in formato *YAML* di CodeBuild, riducendo il lavoro necessario all’adattamento per altri progetti.

L’unico reale problema sorto durante lo sviluppo è stato causato da un bug del **BitBucket Plugin** per Jenkins, che non riusciva ad avviare con successo una pipeline configurata correttamente, poi circumnavigato grazie alla creazione di un secondo processo (Job), avviato dal plugin correttamente, che passava i dati ricevuti alla pipeline corretta.

Conclusioni

Il progetto svolto durante i 3 mesi ha portato un drastico cambiamento nel modo in cui il team di sviluppo del progetto Sphere si approccia alla scrittura, revisione e qualità del codice. Questo miglioramento è molto pronunciato nello sviluppo delle componenti più attive, ovvero i moduli Mobile ed i servizi Backend, dove il testing estensivo e il processo standardizzato han permesso di ottenere prestazioni di sviluppo notevolmente migliorate ed una *awareness* del progetto superiore a prima.

I dati presentati nelle tabelle 3.1 e 3.2 fanno inoltre evincere un miglioramento netto di tutti gli indicatori di performance definiti a discapito di un costo (monetario) di mantenimento superiore a causa di tutte le risorse create per gestire e controllare il processo *DevOps*. L'unico punto **non soddisfatto** tra i vari *tasks* riguardanti il processo di **Continuous Integration**, è stato l'implementazione di un sistema di **Code Coverage** funzionante per tutti i servizi di backend, che però ha un impatto variabile essendo già presenti molteplici *unit* ed *integration* tests.

L'integrazione di un sistema completo di **analisi della Code Coverage** su servizi basati su linguaggio **C++** e **Python**, è sicuramente uno dei primi punti salienti da integrare in futuro, oltre che indispensabile per completare al meglio l'analisi dei test e del codice in se, aggiungendo questi dati all'analisi effettuata con SonarQube.

Una estensione molto interessante potrebbe essere l'introduzione del processo di **Continuous Deployment**, come seguito al processo di *Continuous Delivery*, sfruttando la standardizzazione del deployment che viene effettuato in ambiente Kubernetes. A riguardo potrebbe essere interessante esplorare l'utilizzo di metodologie come **GitOps**[36] (con strumenti come *Argo CD* o *Jenkins X*), che definiscono in modo dichiarativo l'intero sistema, compresi deployment e configurazioni degli stessi, permettendo quindi di integrare un task come una *release* nel classico processo di sviluppo mediante Pull Requests e Code Review.

Il progetto è stato particolarmente *challenging* fin dall'inizio, con l'introduzione di strumenti che personalmente non avevo mai utilizzato ne configurato, ma soprattutto il flusso finale è stato risultato di diversi cambiamenti effettuati in diverse iterazioni dello sviluppo, segno che anche lo sviluppo di un flusso *DevOps-oriented* necessita di un approccio *Agile*.

Molte tecnologie, particolarità delle stesse, complessità nel design, sono state apprese durante il Project Work, segno quindi di una crescita a livello professionale notevole e sicuramente applicabile in futuro in molti ambiti. Altre tecnologie invece, come Jenkins, Docker, SonarQube e in parte l'utilizzo di AWS, le ho potute affinare non poco grazie alla complessità superiore di questo progetto rispetto ad altri precedenti.

Bibliografia

- [1] Cristian Mesiano. *Our endeavour for home digitisation*. URL: <https://www.linkedin.com/pulse/our-endeavour-home-digitisation-cristian-mesiano/>.
- [2] Alexandra Altvater. «What Is SDLC? Understand the Software Development Life Cycle». In: *Stackify* (2020). DOI: <https://stackify.com/what-is-sdlc/>.
- [3] Airbrake. *Waterfall Model: What Is It and When Should You Use It?* URL: <https://airbrake.io/blog/sdlc/waterfall-model>.
- [4] Wikipedia. *Iterative and incremental development*. URL: https://en.wikipedia.org/wiki/Iterative_and_incremental_development.
- [5] Agile Manifesto. *Manifesto per lo Sviluppo Agile di Software*. URL: <https://agilemanifesto.org/iso/it/manifesto.html>.
- [6] Claire Drumond. «<https://www.atlassian.com/agile/scrum>». In: *Atlassian* (). DOI: <https://www.atlassian.com/agile/scrum>.
- [7] K. Beck. «Extreme Programming Explained: Embrace Change». In: (gen. 2000). URL: https://www.researchgate.net/publication/242350998_Extreme_Programming_Explained_Embrace_Change.
- [8] Steve Mezak. «The Origins of DevOps: What's in a Name?» In: *DevOps.com* (2018). DOI: <https://devops.com/the-origins-of-devops-whats-in-a-name/>.
- [9] Atlassian. *CALMS Framework*. URL: <https://www.atlassian.com/devops/frameworks/calms-framework>.
- [10] Rouan Wilsenach. «*DevOpsCulture*». In: *Martin Fowler Website* (2015). DOI: <https://martinfowler.com/bliki/DevOpsCulture.html>.
- [11] Atlassian. *DevOps Trends Survey 2020*. URL: <https://www.atlassian.com/whitepapers/devops-survey-2020>.
- [12] Lucidchart Content Team. *Understanding the DevOps process flow*. URL: <https://www.lucidchart.com/blog/devops-process-flow>.
- [13] Scaled Agile. *Continuous Integration*. URL: <https://www.scaledagileframework.com/continuous-integration/>.
- [14] Red Hat. *Cosa si intende con CI/CD?* URL: <https://www.redhat.com/it/topics/devops/what-is-ci-cd>.
- [15] Scaled Agile. *Continuous Deployment*. URL: <https://www.scaledagileframework.com/continuous-deployment/>.
- [16] CircleCI Puppet. *State of DevOps Report 2020*. Rapp. tecn. Gen. 2021. URL: <https://puppet.com/resources/report/2020-state-of-devops-report/>.

- [17] Christopher Null. *How to measure DevOps ROI*. URL: <https://techbeacon.com/devops/how-measure-devops-roi>.
- [18] Tim Grance Peter Mell. «The NIST Definition of Cloud Computing». In: SP 800-145 (2011). DOI: <https://csrc.nist.gov/publications/detail/sp/800-145/final>.
- [19] Lauren Gibbons Paul Stephanie Overby Lynn Greiner. «What is an SLA? Best practices for service-level agreements». In: *CIO* (lug. 2017). DOI: <https://www.cio.com/article/2438284/outourcing-sla-definitions-and-solutions.html>.
- [20] Carlos Schults. «What Is Infrastructure as Code? How It Works, Best Practices, Tutorials». In: *Stackify* (2019). DOI: <https://stackify.com/what-is-infrastructure-as-code-how-it-works-best-practices-tutorials/>.
- [21] Adam Bertram. «Configuration as Code: What is it and how is it beneficial?» In: *Octopus Blog* (2020). DOI: <https://octopus.com/blog/configuration-as-code-what-is-it-how-is-it-beneficial>.
- [22] Docker. *What is a Container?* URL: <https://www.docker.com/resources/what-container>.
- [23] Evan Baker. «A Comprehensive Container Runtime Comparison». In: *CapitalOne* (2020). DOI: <https://www.capitalone.com/tech/cloud/container-runtime/>.
- [24] Atlassian. *Monorepos in Git*. URL: <https://www.atlassian.com/git/tutorials/monorepos>.
- [25] Atlassian. *What is Git?* URL: <https://www.atlassian.com/git/tutorials/what-is-git>.
- [26] Bazel. *Overview*. URL: <https://docs.bazel.build/versions/master/bazel-overview.html>.
- [27] Phacility. *Phabricator*. URL: <https://www.phacility.com/phabricator/>.
- [28] Martin Heller. «What is Jenkins? The CI server explained». In: *InfoWorld* (mar. 2020). DOI: <https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html>.
- [29] Ionos. *Terraform: cosa si nasconde dietro il software IaC di HashiCorp?* URL: <https://www.ionos.it/digitalguide/server/tools-e-strumenti/cose-terraform/>.
- [30] Reshma Ahmed. «What Is Ansible? – Configuration Management And Automation With Ansible». In: *Edureka!* (2020). DOI: <https://www.edureka.co/blog/what-is-ansible/>.
- [31] Docker. *Docker Overview*. URL: <https://docs.docker.com/get-started/overview/>.
- [32] The MITRE Corporation. *CVE*. URL: <https://cve.mitre.org/>.
- [33] Quay. *Clair*. URL: <https://quay.github.io/clair/>.
- [34] Forum of Incident Response e Inc. Security Teams. *Common Vulnerability Scoring System SIG*. URL: <https://www.first.org/cvss/>.
- [35] NIST. *National Vulnerability Database*. URL: <https://nvd.nist.gov/general>.

- [36] WeaveWorks. *Guide to GitOps*. URL: <https://www.weave.works/technologies/gitops/>.