



UNIVERSITÀ DEGLI STUDI DI MILANO - BICOCCA

**Scuola di Scienze**

**Dipartimento di Informatica, Sistemistica e Comunicazione**

**Corso di laurea in Informatica**

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

**Relatore:** Prof. Mariani Leonardo

**Tutor Aziendale:** Dott. Mesiano Cristian

**Relazione della prova finale di:**

Renzo Simone

Matricola 781616

**Anno Accademico 2020-2021**

## Abstract

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

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

“ *It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change.* ”

---

Charles Darwin,

# Indice

<b>1</b>	<b>Introduzione</b>	<b>4</b>
1.1	Il Contesto Aziendale . . . . .	4
1.1.1	Storia . . . . .	4
1.1.2	Il progetto Sphere . . . . .	4
1.1.3	Profilo personale in azienda . . . . .	4
1.2	Scopo del Project Work . . . . .	4
1.2.1	Obiettivi . . . . .	4
1.2.2	Pianificazione del Lavoro . . . . .	5
1.2.3	Prodotti Finali . . . . .	5
<b>2</b>	<b>Modelli di Sviluppo del Software</b>	<b>6</b>
2.1	Il ciclo di vita del software . . . . .	6
2.1.1	Le fasi del ciclo di vita . . . . .	6
2.1.2	I modelli classici: Waterfall e Iterativo . . . . .	7
2.2	Modelli <i>Agile</i> . . . . .	9
2.2.1	La filosofia <i>Agile</i> . . . . .	9
2.2.2	Un modello di successo: SCRUM Framework . . . . .	9
2.2.3	Un modello veloce: eXtreme Programming . . . . .	9
2.3	Metodologie di Sviluppo <i>Agile</i> : DevOps . . . . .	9
2.3.1	Un metodo ed un'etica . . . . .	9
2.3.2	Obiettivi delle pratiche DevOps . . . . .	9
2.3.3	Processo di Riferimento . . . . .	9
2.4	Il ROI del DevOps . . . . .	9
<b>3</b>	<b>Sviluppo ed Automazione su Cloud</b>	<b>10</b>
3.1	Perchè il Cloud? . . . . .	10
3.2	<i>Infrastructure-as-a-Code</i> . . . . .	10
3.3	<i>Configuration-as-a-Code</i> . . . . .	10
3.4	Containers ed ambienti controllati . . . . .	10
<b>4</b>	<b>Analisi del Processo di Sviluppo</b>	<b>11</b>
4.1	Struttura del Progetto . . . . .	12
4.2	Il processo di sviluppo . . . . .	12
4.3	Requisiti e KPI . . . . .	12
4.4	Il processo DevOps . . . . .	12
4.4.1	Architettura High-Level e Fasi . . . . .	12
4.4.2	Gestione del Codice . . . . .	12
4.4.3	Pull Requests e Code Review . . . . .	12

4.4.4	Continuous Integration . . . . .	12
4.4.5	Continuous Delivery . . . . .	12
4.4.6	Deployment . . . . .	12
4.5	Tecnologie e Strumenti . . . . .	12
4.5.1	SCM: Git . . . . .	12
4.5.2	Build System: Bazel . . . . .	12
4.5.3	Cloud Provider: AWS . . . . .	12
4.5.4	PR Management: Phabricator ed Arcanist . . . . .	12
4.5.5	CI/CD: Jenkins . . . . .	12
4.5.6	Code Analysis: SonarQube . . . . .	12
<b>5</b>	<b>Tecnologie di Background</b>	<b>13</b>
5.1	Cloud Provider: AWS . . . . .	13
5.2	Infrastructure-as-a-Code: <i>Terraform</i> . . . . .	13
5.3	Configuration-as-a-Code: <i>Ansible</i> . . . . .	13
5.4	Container Engine: <i>Docker</i> . . . . .	13
<b>6</b>	<b>Architettura Cloud</b>	<b>14</b>
6.1	Diagramma Architetturale . . . . .	14
6.2	Configurazione di Phabricator . . . . .	14
6.3	Configurazione di Jenkins . . . . .	14
6.3.1	Agent su AWS EC2 . . . . .	14
6.3.2	Agent macOS On-Premise . . . . .	14
6.4	Configurazione di SonarQube . . . . .	14
6.5	Creazione ambiente di build con Docker . . . . .	14
<b>7</b>	<b>La Pipeline di CI</b>	<b>15</b>
7.1	Tecnologie e Strumenti . . . . .	15
7.1.1	Phabricator ed Arcanist: un flusso controllato . . . . .	15
7.1.2	Jenkins: il motore del processo . . . . .	15
7.1.3	Docker: ambiente di testing unificato . . . . .	15
7.1.4	SonarQube: controllo qualità . . . . .	15
7.2	Le fasi della Pipeline . . . . .	15
7.3	Analisi del Codice . . . . .	15
7.3.1	Quality Gates . . . . .	15
7.3.2	Code Coverage . . . . .	15
7.3.3	Risposta a cambiamenti nella qualità . . . . .	15
7.4	Risultati di Testing e QA . . . . .	15
<b>8</b>	<b>La Pipeline di CD</b>	<b>16</b>
8.1	Tecnologie e Strumenti . . . . .	16
8.1.1	Jenkins: il motore del processo . . . . .	16
8.1.2	Docker: ambiente di build unificato . . . . .	16
8.2	Le fasi della Pipeline . . . . .	16
8.3	Analisi delle Vulnerabilità . . . . .	16
8.4	Risultati . . . . .	16

<b>9</b>	<b>Conclusioni</b>	<b>17</b>
9.1	Obiettivi Raggiunti . . . . .	17
9.2	Risultati su Requisiti e KPI . . . . .	17
9.3	Evoluzioni Future . . . . .	17
9.4	Considerazioni Personali . . . . .	17

# Capitolo 1

## Introduzione

### 1.1 Il Contesto Aziendale

#### 1.1.1 Storia

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

#### 1.1.2 Il progetto Sphere

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

#### 1.1.3 Profilo personale in azienda

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

### 1.2 Scopo del Project Work

#### 1.2.1 Obiettivi

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

In particolare si prefigge questi obiettivi:

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

### **1.2.2 Pianificazione del Lavoro**

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

INSERIRE UN GANTT.

### **1.2.3 Prodotti Finali**

I prodotti del project work saranno i seguenti:

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

# Capitolo 2

## Modelli di Sviluppo del Software

### 2.1 Il ciclo di vita del software

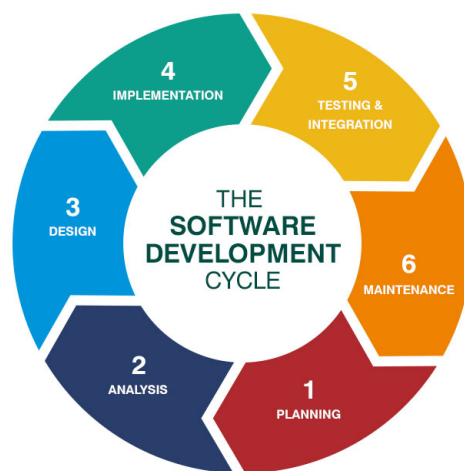


Figura 2.1: Software Development Lifecycle

#### 2.1.1 Le fasi del ciclo di vita

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

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



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

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

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

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

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

## 2.1.2 I modelli classici: Waterfall e Iterativo

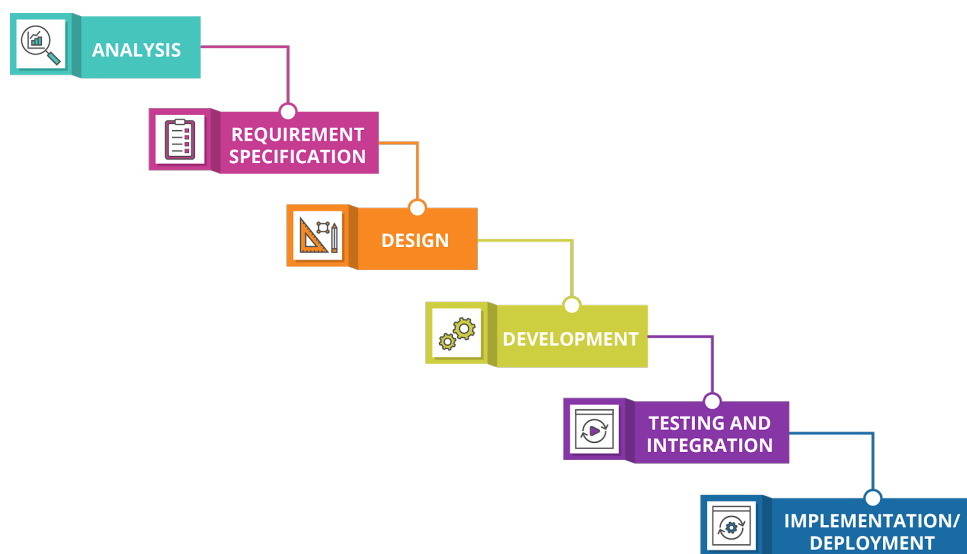


Figura 2.2: Modello Waterfall

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

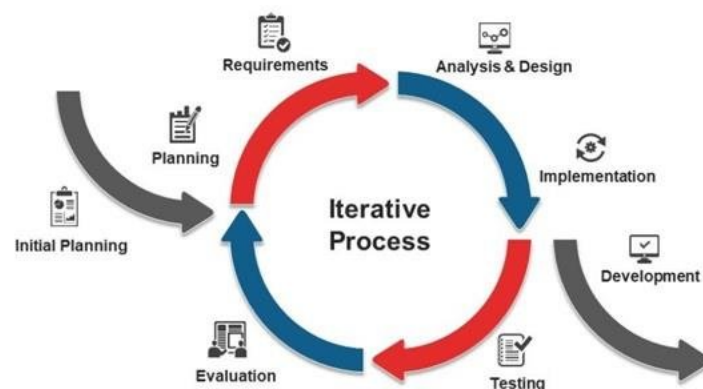


Figura 2.3: Modello Iterativo

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

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

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

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

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

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

## **2.2 Modelli Agile**

### **2.2.1 La filosofia Agile**

### **2.2.2 Un modello di successo: SCRUM Framework**

### **2.2.3 Un modello veloce: eXtreme Programming**

## **2.3 Metodologie di Sviluppo Agile: DevOps**

### **2.3.1 Un metodo ed un'etica**

### **2.3.2 Obiettivi delle pratiche DevOps**

### **2.3.3 Processo di Riferimento**

Continuous Integration

Continuous Delivery

Continuous Deployment

## **2.4 Il ROI del DevOps**

## **Capitolo 3**

# **Sviluppo ed Automazione su Cloud**

### **3.1 Perchè il Cloud?**

### **3.2 *Infrastructure-as-a-Code***

### **3.3 *Configuration-as-a-Code***

### **3.4 Containers ed ambienti controllati**



# **Capitolo 4**

## **Analisi del Processo di Sviluppo**

### **4.1 Struttura del Progetto**

### **4.2 Il processo di sviluppo**

### **4.3 Requisiti e KPI**

### **4.4 Il processo DevOps**

#### **4.4.1 Architettura High-Level e Fasi**

#### **4.4.2 Gestione del Codice**

#### **4.4.3 Pull Requests e Code Review**

#### **4.4.4 Continuous Integration**

Test Automation

Code Analysis

#### **4.4.5 Continuous Delivery**

Build Automation

Artifacts Delivery

#### **4.4.6 Deployment**

### **4.5 Tecnologie e Strumenti**

#### **4.5.1 SCM: Git**

#### **4.5.2 Build System: Bazel**

#### **4.5.3 Cloud Provider: AWS**

#### **4.5.4 PR Management: Phabricator ed Arcanist**

#### **4.5.5 CI/CD: Jenkins**

#### **4.5.6 Code Analysis: SonarQube**

# Capitolo 5

## Tecnologie di Background

**5.1 Cloud Provider: *AWS***

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

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

**5.4 Container Engine: *Docker***

# **Capitolo 6**

## **Architettura Cloud**

### **6.1 Diagramma Architetturale**

### **6.2 Configurazione di Phabricator**

### **6.3 Configurazione di Jenkins**

#### **6.3.1 Agent su AWS EC2**

#### **6.3.2 Agent macOS On-Premise**

### **6.4 Configurazione di SonarQube**

### **6.5 Creazione ambiente di build con Docker**



# **Capitolo 7**

## **La Pipeline di CI**

### **7.1 Tecnologie e Strumenti**

#### **7.1.1 Phabricator ed Arcanist: un flusso controllato**

#### **7.1.2 Jenkins: il motore del processo**

#### **7.1.3 Docker: ambiente di testing unificato**

#### **7.1.4 SonarQube: controllo qualità**

### **7.2 Le fasi della Pipeline**

### **7.3 Analisi del Codice**

#### **7.3.1 Quality Gates**

#### **7.3.2 Code Coverage**

#### **7.3.3 Risposta a cambiamenti nella qualità**

### **7.4 Risultati di Testing e QA**

# **Capitolo 8**

## **La Pipeline di CD**

### **8.1 Tecnologie e Strumenti**

#### **8.1.1 Jenkins: il motore del processo**

#### **8.1.2 Docker: ambiente di build unificato**

### **8.2 Le fasi della Pipeline**

### **8.3 Analisi delle Vulnerabilità**

### **8.4 Risultati**

# **Capitolo 9**

## **Conclusioni**

### **9.1 Obiettivi Raggiunti**

### **9.2 Risultati su Requisiti e KPI**

### **9.3 Evoluzioni Future**

### **9.4 Considerazioni Personali**