Alma Mater Studiorum - Università di Bologna
Dipartimento di Informatica – Scienza e Ingegneria

Corso di Laurea triennale in Informatica per il Management

Tesi in Ingegneria del software

“Adozione delle pratiche DevOps e CI/CD nelle organizzazioni software: un’analisi della letteratura”

Il Relatore:

Chiar.mo Prof. Davide Rossi

Presentata da:

Michele Marini

Anno Accademico 2024/2025

Sommario

Introduzione

Negli ultimi decenni, la trasformazione digitale ha ridefinito il ruolo del software all'interno delle organizzazioni di ogni dimensione e settore. Il software non rappresenta più un semplice strumento di supporto alle attività aziendali, ma è divenuto una componente strategica, spesso centrale, del modello di business. La capacità di sviluppare, rilasciare e mantenere applicazioni software in modo rapido, affidabile e ripetibile costituisce pertanto un fattore critico di competitività, tanto per le grandi imprese quanto per le piccole e medie organizzazioni.

Tradizionalmente, lo sviluppo software è stato organizzato secondo modelli sequenziali o parzialmente iterativi, nei quali le attività di sviluppo e quelle di gestione operativa erano nettamente separate. Questa separazione, storicamente radicata nella struttura organizzativa delle aziende IT, ha spesso generato inefficienze, rallentamenti nei rilasci, difficoltà di integrazione e un aumento del rischio di errori in produzione. Le conseguenze di tale frammentazione si manifestano in modo diverso a seconda del contesto: nelle grandi organizzazioni, la complessità dei processi e la molteplicità dei team coinvolti possono amplificare i tempi di coordinamento; nelle realtà più piccole, la carenza di risorse dedicate e l'assenza di processi formalizzati rendono particolarmente vulnerabile la gestione dei rilasci.

In risposta a tali criticità, negli ultimi anni si è affermato l'approccio **DevOps**, che mira a integrare sviluppo software (Development) e gestione operativa (Operations) attraverso pratiche, strumenti e processi condivisi. DevOps promuove l'automazione del ciclo di vita del software, l'uso sistematico di sistemi di versionamento, pipeline di integrazione e distribuzione continua (CI/CD) e una maggiore collaborazione tra i diversi attori coinvolti nel processo di rilascio.

Nonostante la crescente diffusione di DevOps e delle pratiche CI/CD, la letteratura evidenzia come il livello di adozione sia ancora molto variabile tra le organizzazioni, e come le sfide legate all'implementazione di tali pratiche siano spesso sottovalutate. Molte organizzazioni si trovano in una fase di transizione, in cui gli strumenti sono parzialmente adottati ma i processi non sono ancora completamente automatizzati o standardizzati.

L'obiettivo di questa tesi è analizzare il livello di adozione delle pratiche DevOps e CI/CD all'interno dei team di sviluppo software, con particolare attenzione alla gestione del versionamento del codice e dei rilasci. A tal fine, il lavoro combina una parte teorica, dedicata all'inquadramento dei concetti fondamentali di Ingegneria del Software, DevOps e

CI/CD, con una parte empirica basata su un questionario rivolto a professionisti del settore IT operanti in contesti aziendali eterogenei, dalle startup alle grandi imprese.

La tesi si propone di rispondere alle seguenti domande di ricerca:

- Qual è il livello attuale di adozione delle pratiche DevOps e CI/CD nei team di sviluppo software?
- Quali sono le principali criticità nella gestione del versionamento e dei rilasci software?
- Esiste una relazione tra il livello di maturità CI/CD e la frequenza dei problemi operativi?
- In che misura una gestione strutturata e automatizzata del ciclo di rilascio viene percepita come fattore di miglioramento?
- Quali differenze emergono tra organizzazioni di diversa dimensione nell'adozione di tali pratiche?

Il lavoro è strutturato in cinque capitoli. Il Capitolo 1 introduce il contesto generale del DevOps e del ciclo di vita del software, delineando le criticità dei modelli tradizionali e i principi fondamentali dell'approccio DevOps. Il Capitolo 2 approfondisce i fondamenti teorici, analizzando i modelli di sviluppo, le metodologie Agile, le pratiche CI/CD, la gestione del versionamento e il Release Management. Il Capitolo 3 descrive la metodologia della ricerca empirica condotta, illustrando lo strumento di raccolta dati, la popolazione di riferimento e le tecniche di analisi. Il Capitolo 4 presenta e discute i risultati ottenuti, analizzando il livello di maturità CI/CD, le criticità riscontrate, il confronto tra contesti organizzativi diversi e il valore percepito delle pratiche DevOps. Infine, il Capitolo 5 propone le conclusioni, le implicazioni organizzative e gli sviluppi futuri della ricerca.

Capitolo 1 – DevOps e ciclo di vita del software

1.1 Il ciclo di vita del software (SDLC)

Il **Software Development Life Cycle (SDLC)** rappresenta l'insieme delle fasi attraverso le quali un sistema software viene concepito, sviluppato, distribuito e mantenuto nel tempo [15]. La nozione di ciclo di vita del software si è sviluppata a partire dagli anni Sessanta, in risposta alla cosiddetta "crisi del software", quando la crescente complessità dei sistemi informatici rese evidente la necessità di approcci ingegneristici sistematici alla produzione di software [13].

I modelli di processo software hanno lo scopo di fornire una struttura metodologica per organizzare le attività di sviluppo, migliorandone la prevedibilità, la qualità e la gestibilità. Essi definiscono le fasi del processo, le relazioni tra di esse e i criteri di transizione da una fase alla successiva. Nel corso dei decenni, la disciplina dell'ingegneria del software ha prodotto una pluralità di modelli, ciascuno con specifici punti di forza e limitazioni.

Tra i modelli tradizionali più noti rientra il **modello a cascata (Waterfall)**, proposto da Winston Royce nel 1970 [14], caratterizzato da una sequenza rigida e lineare di fasi: analisi dei requisiti, progettazione, implementazione, test, rilascio e manutenzione. Ogni fase deve essere completata prima di passare alla successiva, e il feedback avviene prevalentemente alla fine del ciclo. Sebbene Royce stesso avesse messo in guardia contro un'applicazione rigida di questo schema, il modello a cascata è stato per lungo tempo il paradigma dominante nello sviluppo software aziendale.

Accanto al modello a cascata si sono sviluppati i modelli **iterativi e incrementali**, tra cui il modello a spirale di Barry Boehm (1988) [4], che introduce cicli ripetuti di pianificazione, analisi del rischio, sviluppo e valutazione. Questi modelli riconoscono che i requisiti evolvono nel tempo e che il software beneficia di cicli ripetuti di sviluppo, verifica e raffinamento. Il modello incrementale, in particolare, prevede che il sistema venga costruito e rilasciato in porzioni successive, ciascuna delle quali aggiunge funzionalità al prodotto.

La comprensione dell'evoluzione dei modelli di sviluppo è essenziale per contestualizzare l'emergere dell'approccio DevOps, che può essere interpretato come il risultato di una progressiva integrazione tra le fasi di sviluppo e quelle di gestione operativa, storicamente trattate come ambiti separati.

Va sottolineato come il passaggio da modelli sequenziali a modelli iterativi e incrementali abbia introdotto una sfida tecnica significativa: la necessità di gestire molteplici versioni del

software in parallelo, di integrare frequentemente le modifiche provenienti da diversi sviluppatori e di automatizzare i processi di verifica e rilascio. Queste sfide tecniche hanno posto le basi per lo sviluppo delle pratiche di Continuous Integration e Continuous Delivery che verranno analizzate nel capitolo successivo [6].

1.2 Criticità dei modelli tradizionali

Una delle principali criticità dei modelli tradizionali di sviluppo software è la netta separazione tra la fase di sviluppo e la fase di esercizio, comunemente indicata come il "muro" tra Development e Operations. In questo paradigma, gli sviluppatori si concentrano sulla produzione del codice e sulla sua verifica funzionale, mentre la gestione degli ambienti, delle build, dei rilasci e del monitoraggio in produzione è demandata ad altre figure, spesso appartenenti a unità organizzative distinte.

Questa separazione genera una serie di problemi ricorrenti e ben documentati nella letteratura. L'integrazione del codice sviluppato da diversi membri del team diventa complessa e rischiosa quando le modifiche vengono consolidate solo occasionalmente, fenomeno noto come "integration hell". I conflitti tra diverse linee di sviluppo possono richiedere ore o giorni di lavoro per essere risolti, con un impatto significativo sulla produttività del team.

Gli errori di configurazione durante il rilascio rappresentano un'altra criticità frequente. Quando il processo di deploy è manuale, le operazioni sono intrinsecamente soggette a imprecisioni: variabili di ambiente non aggiornate, dipendenze mancanti, script di migrazione non eseguiti. Il fenomeno è sintetizzato dall'espressione colloquiale "works on my machine", che evidenzia la discrepanza tra l'ambiente di sviluppo locale e quello di produzione.

I tempi lunghi per il passaggio in produzione derivano dalla necessità di verifiche manuali estese, spesso non standardizzate, e dalla complessità del coordinamento tra team diversi. In molte organizzazioni, il rilascio è un evento raro, complesso e stressante, che richiede il coinvolgimento di molteplici figure e la predisposizione di piani di rollback manuali.

Infine, la scarsa tracciabilità delle versioni rilasciate rende difficoltoso il rollback in caso di malfunzionamenti e complica l'identificazione dell'origine degli errori. L'assenza di una correlazione chiara tra le modifiche apportate al codice e le versioni distribuite in produzione compromette la capacità dell'organizzazione di apprendere dai propri errori e di migliorare progressivamente i propri processi.

Tali problematiche incidono negativamente sulla qualità del software e aumentano il rischio di malfunzionamenti, con conseguenze economiche e organizzative rilevanti. Le interruzioni

di servizio, i bug in produzione e i ritardi nelle consegne hanno un costo che, secondo diverse stime di settore, può essere fino a cento volte superiore al costo della correzione dello stesso difetto se individuato nelle fasi iniziali dello sviluppo.

1.3 L'approccio DevOps

DevOps nasce come risposta strutturale alle inefficienze dei modelli tradizionali, proponendo un approccio orientato alla collaborazione continua tra sviluppo e operations. Il termine fu coniato da Patrick Debois nel 2009, in occasione della prima conferenza *DevOpsDays* tenutasi a Gent, in Belgio [18]. Da allora, DevOps si è rapidamente affermato come uno dei paradigmi più influenti nel panorama dell'ingegneria del software contemporanea.

DevOps non identifica una metodologia rigida o un framework prescrittivo, ma piuttosto un insieme di **principi**, **pratiche** e **strumenti** finalizzati a migliorare l'intero ciclo di vita del software. Il modello CALMS, proposto da Jez Humble [11], sintetizza i pilastri fondamentali di DevOps in cinque dimensioni:

- **Culture:** la cultura della collaborazione e della responsabilità condivisa tra team di sviluppo e team operativi, con l'obiettivo di abbattere i silos organizzativi;
- **Automation:** l'automazione dei processi ripetitivi, dalla build al test al deploy, per ridurre gli errori umani e accelerare il ciclo di rilascio;
- **Lean:** l'applicazione dei principi lean alla gestione del flusso di lavoro, con focus sulla riduzione degli sprechi e sull'ottimizzazione del flusso di valore;
- **Measurement:** la misurazione continua delle prestazioni attraverso metriche oggettive, per guidare le decisioni di miglioramento;
- **Sharing:** la condivisione della conoscenza, degli strumenti e delle responsabilità tra tutti gli attori coinvolti.

Questi principi si traducono in un cambiamento culturale prima ancora che tecnologico. L'adozione di DevOps richiede l'abbattimento delle barriere organizzative tra i team, la promozione di una mentalità orientata al miglioramento continuo e l'accettazione del fallimento come opportunità di apprendimento [11].

Dal punto di vista tecnico, DevOps fa ampio uso di strumenti per il versionamento del codice (come Git), la continuous integration (come Jenkins, GitLab CI, GitHub Actions), la continuous delivery e deployment, la containerizzazione (Docker, Kubernetes), la gestione delle infrastrutture come codice (Terraform, Ansible) e il monitoraggio dei sistemi (Prometheus, Grafana, ELK Stack). L'ecosistema degli strumenti DevOps è vasto e in continua evoluzione, e la scelta degli strumenti più adeguati dipende dal contesto specifico dell'organizzazione.

1.4 DevOps e qualità del software

L'adozione delle pratiche DevOps ha un impatto diretto e misurabile sulla **qualità del software**, intesa sia come qualità del prodotto finale sia come qualità del processo di sviluppo. L'automazione delle build e dei test consente di individuare tempestivamente errori e regressioni, riducendo il costo di correzione secondo il principio dello "shift-left testing", che prevede lo spostamento delle attività di verifica nelle fasi più precoci del ciclo di sviluppo.

La standardizzazione delle pipeline di rilascio riduce il rischio di configurazioni incoerenti tra ambienti diversi, garantendo che il software venga costruito, testato e distribuito in modo identico e ripetibile. L'uso di ambienti di staging, la pratica del deploy automatizzato e le strategie di rilascio progressivo (canary release, blue-green deployment, feature flags) consentono di minimizzare l'impatto dei difetti che inevitabilmente sfuggono ai test.

Inoltre, l'uso sistematico di sistemi di versionamento e di strategie di branching ben definite migliora la tracciabilità delle modifiche e facilita la collaborazione tra sviluppatori. La possibilità di associare ogni modifica a un ticket, una user story o un bug report crea una catena di tracciabilità che si estende dall'ideazione alla produzione.

La letteratura scientifica e professionale conferma ampiamente i benefici dell'adozione DevOps. Il rapporto annuale *State of DevOps*, pubblicato da DORA (DevOps Research and Assessment), basato su migliaia di risposte raccolte a livello globale, ha dimostrato una correlazione significativa tra la maturità delle pratiche DevOps e quattro metriche chiave di performance:

- **Deployment Frequency:** la frequenza con cui il software viene rilasciato in produzione;
- **Lead Time for Changes:** il tempo che intercorre tra il commit di una modifica e il suo rilascio in produzione;
- **Change Failure Rate:** la percentuale di rilasci che causano un fallimento o un degradamento del servizio;
- **Time to Restore Service:** il tempo necessario per ripristinare il servizio dopo un incidente.

Le organizzazioni classificate come "elite performers" nel rapporto DORA [7] rilasciano software su richiesta (anche più volte al giorno), con un lead time inferiore a un'ora, un change failure rate compreso tra lo 0% e il 15% e un tempo di ripristino inferiore a un'ora. Questi dati evidenziano il potenziale trasformativo dell'adozione sistematica di pratiche DevOps e CI/CD.

Alla luce di queste considerazioni, DevOps rappresenta un approccio in grado di migliorare in modo sostanziale la gestione del ciclo di vita del software, tema che verrà approfondito nel capitolo successivo attraverso l'analisi dei fondamenti teorici e successivamente nell'analisi empirica.

Capitolo 2 – Fondamenti teorici e contesto di riferimento

2.1 Ingegneria del software e gestione del ciclo di vita

L'ingegneria del software, come disciplina accademica e professionale, studia metodologie, strumenti e processi finalizzati alla progettazione, sviluppo, manutenzione e gestione del software lungo l'intero ciclo di vita. Essa si è sviluppata a partire dalla conferenza NATO del 1968 a Garmisch, in Germania, che per prima utilizzò il termine "software engineering" per descrivere l'applicazione di principi ingegneristici alla produzione di software.

Nel contesto aziendale moderno, il software rappresenta una risorsa strategica il cui valore non dipende esclusivamente dalla qualità del codice, ma anche dall'efficacia dei processi organizzativi che ne regolano lo sviluppo e il rilascio. La gestione del ciclo di vita del software (SDLC) comprende tutte le attività necessarie per portare un prodotto software dall'ideazione alla dismissione: pianificazione, analisi dei requisiti, progettazione architetturale e di dettaglio, sviluppo, test, rilascio, manutenzione evolutiva e correttiva, e infine ritiro [13].

La qualità complessiva di un prodotto software dipende dalla qualità di ciascuna di queste fasi e, soprattutto, dalla qualità delle transizioni tra una fase e l'altra. È proprio nelle interfacce tra le fasi che si annidano molte delle inefficienze e degli errori più costosi, rendendo cruciale la definizione di processi chiari e possibilmente automatizzati per gestire tali transizioni.

2.2 Modelli di sviluppo del software

Il **modello a cascata (Waterfall)** è stato storicamente il primo modello formalizzato per la gestione del ciclo di sviluppo software. Proposto da Winston Royce nel 1970 [14], esso organizza il processo in fasi sequenziali rigide. Nonostante la sua semplicità concettuale e la chiara definizione delle responsabilità per ciascuna fase, il modello a cascata presenta limiti significativi: la rigidità della sequenza rende difficile gestire i cambiamenti di requisiti, il feedback dell'utente arriva solo alla fine del ciclo, e il costo di correzione degli errori cresce esponenzialmente quanto più tardi vengono individuati.

Il **modello a spirale** di Barry Boehm (1988) [4] rappresenta un'evoluzione significativa, introducendo il concetto di analisi del rischio come elemento strutturale del processo. Il

modello organizza lo sviluppo in cicli iterativi, ciascuno dei quali comprende pianificazione, analisi del rischio, sviluppo e valutazione. Questo approccio consente di affrontare i rischi maggiori nelle fasi iniziali del progetto e di adattare il piano di sviluppo sulla base delle evidenze raccolte.

I modelli iterativi e incrementali condividono il principio fondamentale di suddividere il progetto in cicli brevi, ciascuno dei quali produce un incremento funzionante del software. Questo approccio riduce il rischio di fallimento complessivo del progetto, consente di raccogliere feedback dagli utenti in modo continuativo e facilita l'adattamento ai cambiamenti.

Tuttavia, anche i modelli iterativi presentano sfide nella pratica, in particolare per quanto riguarda la gestione della configurazione, il versionamento delle diverse iterazioni e il coordinamento dei rilasci. Queste sfide hanno contribuito allo sviluppo delle pratiche di Continuous Integration e Continuous Delivery.

2.3 Metodologie Agile

Le metodologie Agile, formalizzate nel 2001 con la pubblicazione del Manifesto Agile [17], rappresentano un cambio di paradigma nello sviluppo software. I quattro valori fondamentali del manifesto privilegiano: gli individui e le interazioni rispetto ai processi e agli strumenti; il software funzionante rispetto alla documentazione esaustiva; la collaborazione con il cliente rispetto alla negoziazione contrattuale; la risposta al cambiamento rispetto al rispetto di un piano prestabilito.

Tra le implementazioni più diffuse si annoverano **Scrum** e **Kanban**. Scrum [22] organizza il lavoro in iterazioni chiamate sprint, della durata tipica di due-quattro settimane, al termine delle quali viene prodotto un incremento potenzialmente rilasciabile del prodotto. Il framework definisce ruoli specifici (Product Owner, Scrum Master, Development Team), artefatti (Product Backlog, Sprint Backlog, Increment) e cerimonie (Sprint Planning, Daily Scrum, Sprint Review, Sprint Retrospective).

Kanban [1], ispirato al sistema di produzione Toyota, si concentra sulla visualizzazione del flusso di lavoro e sulla limitazione del lavoro in corso (Work In Progress, WIP). Attraverso la Kanban board, il team visualizza lo stato di avanzamento di ciascun elemento di lavoro, identifica i colli di bottiglia e ottimizza il flusso continuo delle attività.

Un aspetto spesso sottovalutato delle metodologie Agile è la loro esigenza implicita di infrastruttura tecnica solida. La promessa di rilasci frequenti e incrementali richiede la capacità di integrare, testare e distribuire rapidamente le modifiche, operazioni che diventano insostenibili se svolte manualmente. Questa esigenza ha creato le precondizioni

tecniche e culturali per lo sviluppo delle pratiche di Continuous Integration e Continuous Delivery.

Le metodologie Agile hanno preparato il terreno culturale per l'adozione di DevOps, promuovendo principi come la collaborazione, il feedback rapido, il rilascio frequente e l'adattamento continuo. DevOps può essere interpretato come l'estensione dei principi Agile oltre il confine del team di sviluppo, includendo le operazioni, il rilascio e il monitoraggio nel ciclo iterativo. Se Agile ha abbattuto il muro tra business e sviluppo, DevOps mira ad abbattere il muro tra sviluppo e operations.

2.4 Continuous Integration (CI)

La **Continuous Integration (CI)** è una pratica di sviluppo software che prevede l'integrazione frequente del codice sorgente all'interno di un repository condiviso. Introdotta da Kent Beck nell'ambito dell'Extreme Programming e successivamente approfondita da Martin Fowler [3] [20], la CI si basa sul principio che integrare piccole modifiche con alta frequenza è meno rischioso e meno costoso rispetto a integrazioni rare e massicce.

Ogni integrazione attiva una pipeline automatizzata che include la compilazione del software (build), l'esecuzione di test automatici a diversi livelli (unit test, integration test, end-to-end test) e la verifica della qualità del codice attraverso strumenti di analisi statica [6]. L'obiettivo della CI è individuare errori il prima possibile, riducendo il costo di correzione e migliorando la stabilità del sistema.

La CI richiede alcune precondizioni organizzative e tecniche: un sistema di versionamento condiviso, una suite di test automatici con copertura adeguata, un server di build in grado di eseguire la pipeline ad ogni commit e una cultura del team orientata all'integrazione frequente. Strumenti come Jenkins, GitLab CI/CD, GitHub Actions, CircleCI e Azure DevOps sono tra i più utilizzati per implementare pipeline di CI.

Tra le pratiche fondamentali della CI si annoverano: il commit frequente sul repository condiviso (idealmente più volte al giorno per ciascuno sviluppatore), la risoluzione immediata dei build falliti, la manutenzione di una suite di test rapida e affidabile, e la visibilità costante dello stato della build per tutto il team.

2.5 Continuous Delivery e Continuous Deployment

La **Continuous Delivery (CD)** estende la CI introducendo la possibilità di rilasciare il software in produzione in modo ripetibile e affidabile, mantenendo il sistema sempre in uno stato potenzialmente distribuibile. Come descritto da Humble e Farley (2010) [9], l'idea centrale è che il team di sviluppo possa rilasciare una nuova versione del software in

qualsiasi momento con un livello di rischio contenuto, grazie all'automazione completa della pipeline.

La **Continuous Deployment** rappresenta un'evoluzione ulteriore, in cui ogni modifica che supera i controlli automatici viene rilasciata automaticamente in produzione senza intervento umano. Mentre la Continuous Delivery lascia all'uomo la decisione finale sul rilascio, la Continuous Deployment elimina anche quest'ultimo passaggio manuale.

La distinzione tra Continuous Delivery e Continuous Deployment è rilevante nella pratica. La Continuous Delivery è applicabile in qualsiasi contesto, inclusi quelli regolamentati. La Continuous Deployment richiede un livello di maturità molto elevato dei test automatici e una cultura organizzativa che accetti il rilascio automatico. La scelta tra le due modalità dipende dal contesto specifico dell'organizzazione.

Indipendentemente dalla scelta tra Delivery e Deployment, entrambe le modalità beneficiano di strategie di rilascio progressivo che consentono di ridurre il rischio associato a ogni singolo rilascio. Il canary release prevede il rilascio della nuova versione a una percentuale limitata di utenti, monitorando le metriche chiave prima di estendere il rilascio all'intera base utenti. Il blue-green deployment mantiene due ambienti di produzione identici, consentendo di indirizzare il traffico dall'uno all'altro istantaneamente e di eseguire un rollback immediato in caso di problemi. Le feature flags consentono di rilasciare codice contenente funzionalità non ancora attivate, separando il rilascio tecnico dall'attivazione funzionale [9].

2.6 Gestione del versionamento del codice

Il versionamento del codice è una componente fondamentale dei moderni processi di sviluppo. **Git**, creato da Linus Torvalds nel 2005, è diventato lo standard de facto per il controllo di versione. Si tratta di un sistema distribuito in cui ogni sviluppatore dispone di una copia completa della repository, comprensiva dell'intera storia delle modifiche.

Le strategie di branching più diffuse includono diverse opzioni. Git Flow, proposto da Vincent Driessen nel 2010 [19], prevede branch separati per feature, release e hotfix, organizzati attorno a due branch principali (main e develop). GitHub Flow, più semplice, è basato su un singolo branch principale con feature branch di breve durata e pull request. Trunk-Based Development prevede commit frequenti direttamente sul branch principale, con eventuale uso di feature flags.

La scelta della strategia di branching incide significativamente sulla velocità di integrazione, sulla complessità della gestione dei conflitti e sulla frequenza dei rilasci. Team di piccole dimensioni tendono a beneficiare di strategie più semplici come GitHub Flow e Trunk-Based

Development, mentre team più grandi o progetti con requisiti di release management complessi possono preferire strategie più strutturate come Git Flow. La coerenza tra la strategia di branching e la pipeline CI/CD è fondamentale: una strategia di branching complessa richiede pipeline in grado di gestire build e test su molteplici branch, aumentando la complessità dell'infrastruttura.

Le piattaforme di hosting come GitHub, GitLab e Bitbucket integrano il versionamento con funzionalità di collaborazione (pull request, code review), gestione dei progetti (issue tracking, board), pipeline CI/CD e monitoraggio, creando ecosistemi completi per la gestione del ciclo di vita del software. La scelta della piattaforma rappresenta una decisione strategica che influenza la produttività del team e la qualità dei processi.

2.7 Release Management

Il **Release Management** comprende l'insieme di attività organizzative e tecniche finalizzate a pianificare, coordinare e controllare il rilascio del software negli ambienti di destinazione. Il suo obiettivo è garantire che le nuove versioni vengano distribuite in modo controllato, minimizzando il rischio di interruzioni [9].

In contesti poco strutturati, il release management è spesso svolto in modo informale, senza processi documentati né strumenti dedicati. Questa informalità aumenta il rischio di errori durante il deploy, rende difficoltosa la tracciabilità e complica il rollback.

L'introduzione di processi strutturati consente di trasformare il rilascio da evento eccezionale a routine quotidiana. Tra le pratiche più efficaci: pipeline di deployment automatizzato, strategie di rilascio progressivo (canary release, blue-green deployment), ambienti di staging per la validazione pre-produzione, e il semantic versioning (SemVer) per la numerazione delle versioni.

2.8 DevOps: sfide organizzative nell'adozione

L'adozione di DevOps non è priva di sfide. Le organizzazioni di grandi dimensioni devono affrontare la complessità della trasformazione culturale, la necessità di coordinare molteplici team e la gestione di architetture legacy spesso monolitiche e scarsamente adatte all'automazione. La migrazione verso architetture più moderne, come i microservizi, è un processo lungo, costoso e rischioso che richiede competenze specifiche e un investimento significativo.

Le organizzazioni di dimensioni più ridotte, pur beneficiando di una maggiore agilità organizzativa, si confrontano con vincoli differenti: risorse limitate, assenza di figure specializzate e processi informali che rendono difficile l'adozione sistematica di pratiche

strutturate. In questi contesti, le attività DevOps ricadono spesso su sviluppatori che già ricoprono molteplici ruoli, generando un sovraccarico che può compromettere sia la qualità del lavoro di sviluppo sia quella della gestione infrastrutturale.

La containerizzazione con Docker e l'orchestrazione con Kubernetes [5] hanno accelerato significativamente l'adozione di DevOps, risolvendo alla radice il problema della discrepanza tra ambienti. L'Infrastructure as Code (IaC), implementata con strumenti come Terraform e Ansible, consente di gestire l'infrastruttura attraverso file di configurazione versionabili, testabili e riproducibili, applicando le stesse best practice utilizzate per il codice applicativo.

Un aspetto particolarmente rilevante è il ruolo della cultura organizzativa. Ron Westrum ha proposto una tipologia di culture organizzative [16], adottata dalla ricerca DORA [7], che distingue tra culture patologiche (orientate al potere), burocratiche (orientate alle regole) e generative (orientate alla performance). Le evidenze mostrano che le organizzazioni con cultura generativa, caratterizzate da alta collaborazione e condivisione delle responsabilità, ottengono prestazioni DevOps significativamente superiori [7]. Questo suggerisce che il cambiamento culturale è un prerequisito, non una conseguenza, dell'adozione efficace di DevOps.

Indipendentemente dalla dimensione dell'organizzazione, la letteratura e la pratica convergono nell'indicare l'automazione dei rilasci come un fattore abilitante per migliorare efficienza e qualità [7] [11].

Capitolo 3 – Strumenti e architetture per il ciclo di rilascio

3.1 L'ecosistema degli strumenti DevOps

L'ecosistema degli strumenti DevOps si è sviluppato rapidamente nell'ultimo decennio, passando da soluzioni isolate a piattaforme integrate che coprono l'intero ciclo di vita del software. La scelta degli strumenti rappresenta una decisione strategica per le organizzazioni, in quanto influenza la produttività dei team, la qualità dei rilasci e la capacità di evoluzione dei processi nel tempo [2].

L'ecosistema può essere suddiviso in categorie funzionali principali: sistemi di controllo versione, piattaforme di CI/CD, strumenti di containerizzazione e orchestrazione, soluzioni di Infrastructure as Code, strumenti di monitoraggio e osservabilità, e piattaforme di collaborazione. Queste categorie non sono mutuamente esclusive: le piattaforme moderne tendono a integrare funzionalità trasversali, offrendo un'esperienza sempre più unificata che riduce la frammentazione e semplifica la governance.

La tendenza verso l'integrazione è particolarmente evidente nelle piattaforme di nuova generazione come GitLab e GitHub, che combinano in un'unica soluzione il repository del codice, le pipeline CI/CD, la gestione dei progetti, il registry delle immagini container e gli strumenti di sicurezza. Questa convergenza riflette una consapevolezza crescente nel settore: la frammentazione degli strumenti genera overhead di gestione, rallenta l'adozione e complica la formazione del personale.

3.2 Piattaforme di CI/CD

Jenkins è storicamente la piattaforma di CI/CD più diffusa, nata nel 2011 come fork di Hudson. La sua architettura basata su plugin offre una flessibilità estrema ma richiede una manutenzione significativa. Jenkins supporta pipeline dichiarative e scripted, definite tramite Jenkinsfile, e può essere eseguito su qualsiasi infrastruttura. Il suo punto di forza è l'enorme ecosistema di plugin, con oltre 1800 estensioni disponibili; il suo limite principale è la complessità di configurazione e manutenzione, che richiede competenze dedicate e tempo di gestione non trascurabile.

GitLab CI/CD rappresenta una soluzione integrata all'interno della piattaforma GitLab, che unisce repository, CI/CD, issue tracking e registry dei container in un'unica applicazione.

Le pipeline sono definite tramite un file .gitlab-ci.yml nella root del repository, un approccio noto come "pipeline as code" che garantisce il versionamento della configurazione insieme al codice sorgente. GitLab offre sia una versione cloud (GitLab.com) sia una versione self-hosted, consentendo alle organizzazioni di scegliere il modello più adatto alle proprie esigenze di compliance e sicurezza.

GitHub Actions è la soluzione CI/CD nativa della piattaforma GitHub, introdotta nel 2019. Basata su workflow definiti in file YAML, offre un marketplace di azioni riutilizzabili create dalla comunità che semplifica notevolmente la costruzione delle pipeline. La sua integrazione nativa con l'ecosistema GitHub (pull request, issues, packages, security advisories) la rende particolarmente efficace per i progetti ospitati su questa piattaforma e ne ha favorito una rapida diffusione, soprattutto nei team di piccole e medie dimensioni.

Altre soluzioni rilevanti nel panorama includono **Azure DevOps** (soluzione Microsoft che integra board, repository, pipeline e gestione dei test), **CircleCI** (piattaforma cloud-native orientata alla velocità di esecuzione) e **Bitbucket Pipelines** (integrata nell'ecosistema Atlassian e particolarmente adatta ai team che già utilizzano Jira e Confluence).

La scelta della piattaforma dipende da molteplici fattori: l'ecosistema preesistente dell'organizzazione, le competenze del team, i requisiti di sicurezza e compliance, il budget disponibile e la complessità delle pipeline necessarie. Un aspetto cruciale è il modello di pricing: le piattaforme cloud-based offrono tier gratuiti con limiti di utilizzo che possono essere sufficienti per team di piccole dimensioni, rendendo l'adozione iniziale priva di costi. Le soluzioni self-hosted come Jenkins non hanno costi di licenza ma richiedono investimenti in infrastruttura e competenze di gestione.

3.3 Containerizzazione e orchestrazione

La **containerizzazione** rappresenta una delle innovazioni tecnologiche più significative degli ultimi anni per l'evoluzione delle pratiche DevOps. **Docker**, rilasciato nel 2013, ha reso la containerizzazione accessibile e pratica, consentendo di impacchettare un'applicazione insieme a tutte le sue dipendenze in un'unità portatile e riproducibile chiamata container.

I container risolvono il problema fondamentale della discrepanza tra ambienti: un container che funziona correttamente nell'ambiente di sviluppo funzionerà in modo identico in test, staging e produzione. Questo principio elimina alla radice una delle cause più frequenti di errori in produzione, quella espressa dall'espressione colloquiale "works on my machine", e semplifica drasticamente il processo di rilascio.

A differenza delle macchine virtuali tradizionali, i container condividono il kernel del sistema operativo host, risultando significativamente più leggeri in termini di risorse (memoria, disco, tempo di avvio). Un container può essere avviato in pochi secondi, contro i minuti necessari per una macchina virtuale, rendendo praticabile la creazione di ambienti effimeri per ogni esecuzione della pipeline CI/CD.

Kubernetes, originariamente sviluppato da Google e rilasciato come progetto open source nel 2014 [5], è diventato lo standard de facto per l'orchestrazione dei container. Kubernetes gestisce il deployment, lo scaling e le operazioni di applicazioni containerizzate su cluster di macchine, fornendo meccanismi di auto-healing (riavvio automatico dei container falliti), load balancing, service discovery e gestione delle configurazioni.

L'adozione di Kubernetes è particolarmente diffusa nelle grandi organizzazioni e nei contesti cloud-native, mentre le realtà più piccole possono optare per soluzioni più semplici come Docker Compose per ambienti di sviluppo e staging, delegando la gestione in produzione a servizi gestiti dai cloud provider come Amazon ECS, Google Cloud Run o Azure Container Instances. Questa graduazione nelle soluzioni di orchestrazione consente a ciascuna organizzazione di adottare il livello di complessità più adatto alle proprie esigenze e capacità.

L'architettura a microservizi, spesso associata alla containerizzazione, organizza un'applicazione come un insieme di servizi indipendenti, ciascuno sviluppabile e rilasciabile autonomamente. Tuttavia, è importante riconoscere che i microservizi introducono complessità aggiuntive: networking tra servizi, gestione della consistenza dei dati distribuiti, debugging trasversale. Per molte organizzazioni, un'architettura monolitica ben strutturata può rappresentare un punto di partenza più pragmatico.

3.4 Infrastructure as Code (IaC)

L'**Infrastructure as Code (IaC)** è la pratica di gestire e provisioning l'infrastruttura IT attraverso file di configurazione leggibili dalla macchina [12], anziché attraverso processi manuali o interfacce grafiche. L'IaC rappresenta un pilastro fondamentale dell'approccio DevOps, in quanto consente di applicare le stesse pratiche di versionamento, testing e revisione utilizzate per il codice applicativo anche all'infrastruttura.

Gli strumenti di IaC possono essere classificati in due categorie principali. Gli strumenti di **provisioning**, come **Terraform** (HashiCorp) e **CloudFormation** (AWS), gestiscono la creazione e configurazione delle risorse infrastrutturali (server, reti, database, servizi cloud). Gli strumenti di **configuration management**, come **Ansible** (Red Hat) e Puppet, gestiscono la configurazione del software sulle macchine già esistenti (installazione di pacchetti, configurazione di servizi, gestione degli utenti).

Terraform si è affermato come standard del settore grazie al suo approccio dichiarativo e alla capacità di gestire risorse su molteplici cloud provider attraverso un linguaggio unificato (HCL, HashiCorp Configuration Language). L'approccio dichiarativo consente di descrivere lo stato desiderato dell'infrastruttura, lasciando allo strumento il compito di determinare le azioni necessarie per raggiungere tale stato, riducendo il rischio di errori e garantendo l'idempotenza delle operazioni.

I benefici dell'IaC sono molteplici: riproducibilità garantita degli ambienti, possibilità di effettuare code review sulle modifiche infrastrutturali, tracciabilità completa delle evoluzioni, capacità di ricreare ambienti identici in pochi minuti e riduzione drastica degli errori di configurazione manuale. In un contesto DevOps maturo, l'infrastruttura è trattata con lo stesso rigore del codice applicativo: versionata, testata, revisionata e rilasciata attraverso pipeline automatizzate.

3.5 Anatomia di una pipeline CI/CD

Una pipeline CI/CD completa è tipicamente composta da una sequenza di stage [6] [9], ciascuno dei quali esegue un insieme di operazioni e produce artefatti o feedback. Sebbene la struttura specifica vari in base al contesto, è possibile identificare gli stage fondamentali che caratterizzano la maggior parte delle implementazioni.

Lo stage di **Source** è attivato da un evento nel repository (commit, pull request, tag) e recupera il codice sorgente. Lo stage di **Build** compila il codice, risolve le dipendenze e produce gli artefatti eseguibili. Lo stage di **Test** esegue test automatici a diversi livelli: unit test per la verifica delle singole componenti, integration test per la verifica delle interazioni tra componenti, end-to-end test per la verifica dei flussi utente completi e, eventualmente, test di performance e sicurezza.

Lo stage di **Deploy to Staging** rilascia gli artefatti in un ambiente che replica le condizioni di produzione per la validazione finale. Lo stage di **Deploy to Production** esegue il rilascio effettivo, tipicamente utilizzando strategie progressive come canary release o blue-green deployment. Infine, lo stage di **Monitoring** verifica lo stato del sistema dopo il rilascio attraverso health check, metriche e log aggregati.

Ogni stage funge da gate di qualità: se un'operazione fallisce, la pipeline si interrompe e il team viene notificato, impedendo la propagazione dell'errore alle fasi successive. Questo meccanismo di "fail fast" è fondamentale per mantenere la qualità e la velocità del processo di rilascio.

Un aspetto spesso sottovalutato nella progettazione delle pipeline è la velocità di esecuzione. Una pipeline lenta scoraggia i commit frequenti e riduce la produttività del team. Le best

practice includono l'esecuzione parallela degli stage indipendenti, l'uso di meccanismi di caching per dipendenze e artefatti, la suddivisione della suite di test in livelli con tempi di esecuzione crescenti e l'uso di build incrementali che ricompilano solo i componenti modificati.

La configurazione della pipeline come codice (pipeline as code), archiviata nel repository insieme al codice applicativo, garantisce che le modifiche alla pipeline seguano lo stesso processo di revisione e approvazione del codice, migliorando la governance e la tracciabilità.

La gestione dei segreti all'interno delle pipeline rappresenta un'area critica dal punto di vista della sicurezza. Credenziali, chiavi API, certificati e altre informazioni sensibili non devono mai essere inclusi nel codice sorgente o nei log della pipeline. Le piattaforme CI/CD moderne offrono meccanismi di gestione dei segreti integrati, spesso con supporto per vault esterni come HashiCorp Vault, che consentono di iniettare le credenziali nell'ambiente di esecuzione in modo sicuro e auditabile.

3.6 Monitoraggio e osservabilità

Il monitoraggio e l'osservabilità rappresentano la componente che chiude il ciclo di feedback DevOps. Mentre il monitoraggio tradizionale si concentra sulla verifica di metriche predefinite (uptime, utilizzo CPU, memoria), l'osservabilità è un concetto più ampio che riguarda la capacità di comprendere lo stato interno di un sistema a partire dai suoi output esterni.

I tre pilastri dell'osservabilità sono le metriche (dati numerici aggregati nel tempo, come latenza, throughput, error rate), i log (registrazioni testuali degli eventi, con diversi livelli di severità) e le tracce distribuite (ricostruzione del percorso di una richiesta attraverso i diversi servizi del sistema). Strumenti come Prometheus e Grafana per le metriche, lo stack ELK (Elasticsearch, Logstash, Kibana) per i log, e Jaeger per il tracing distribuito sono ampiamente adottati nel panorama DevOps.

L'integrazione del monitoraggio con le pipeline CI/CD consente di implementare strategie di rilascio intelligenti, come il rollback automatico in caso di degradamento delle metriche chiave dopo un deploy, o la promozione automatica di una release canary a release completa quando le metriche confermano la stabilità. Questa integrazione trasforma il monitoraggio da attività passiva a componente attiva del processo di rilascio, chiudendo il cerchio dell'automazione.

3.7 Sicurezza nel ciclo DevOps (DevSecOps)

L'integrazione della sicurezza nel ciclo DevOps, nota come **DevSecOps**, rappresenta un'evoluzione del paradigma che riconosce la sicurezza come responsabilità condivisa e come attività da integrare in ogni fase del ciclo di vita [11], anziché delegarla a verifiche finali e spesso tardive.

Le pratiche DevSecOps includono la scansione automatica delle vulnerabilità nelle dipendenze (Software Composition Analysis, SCA), l'analisi statica del codice per l'identificazione di pattern insicuri (Static Application Security Testing, SAST), i test dinamici di sicurezza sulle applicazioni in esecuzione (Dynamic Application Security Testing, DAST), la scansione delle immagini container per vulnerabilità note, e la gestione sicura dei segreti (credenziali, chiavi API, certificati) attraverso vault dedicati come HashiCorp Vault o AWS Secrets Manager.

L'approccio "security as code" consente di definire le policy di sicurezza in file versionabili e di automatizzarne la verifica all'interno della pipeline. Questo approccio è particolarmente rilevante nei contesti regolamentati (finanza, sanità, pubblica amministrazione), dove la conformità alle normative deve essere dimostrabile e auditabile. L'integrazione della sicurezza nella pipeline CI/CD consente di individuare e correggere le vulnerabilità nelle fasi precoci dello sviluppo, riducendo significativamente il costo di remediation.

Capitolo 4 – Adozione di DevOps: evidenze empiriche e sfide organizzative

4.1 Stato dell'adozione di DevOps nelle organizzazioni

L'adozione di DevOps è cresciuta in modo significativo nell'ultimo decennio, ma il livello di maturità raggiunto dalle organizzazioni varia enormemente. I rapporti annuali pubblicati da DORA (DevOps Research and Assessment), le indagini di Puppet Labs e le survey di GitLab e JetBrains forniscono un quadro dettagliato e basato su evidenze [8] [21] dello stato dell'adozione a livello globale.

Le evidenze disponibili indicano che la maggior parte delle organizzazioni si colloca in una fase intermedia di adozione, caratterizzata dall'utilizzo di alcuni strumenti di automazione ma da processi non ancora completamente maturi. Le organizzazioni classificate come "elite performers" dal rapporto DORA rappresentano ancora una minoranza, stimata intorno al 18-20% del campione globale, mentre una quota significativa, circa il 20-25%, opera con pratiche prevalentemente manuali o con automazione parziale.

La distribuzione dei livelli di maturità non è uniforme tra settori industriali, aree geografiche e dimensioni aziendali. Le aziende tecnologiche e le organizzazioni digital-native tendono a presentare livelli di maturità superiori, beneficiando di una cultura organizzativa orientata all'innovazione e di team con competenze specializzate. I settori regolamentati (finanza, sanità, pubblica amministrazione) presentano spesso livelli inferiori, in parte a causa di vincoli di compliance che rallentano l'adozione di pratiche completamente automatizzate.

Nel contesto europeo, e in particolare in quello italiano, il tessuto imprenditoriale è caratterizzato da una forte prevalenza di piccole e medie imprese, molte delle quali hanno avviato processi di digitalizzazione solo negli ultimi anni. In questo contesto, l'adozione di DevOps si confronta con specificità culturali e organizzative: la tendenza a concentrare molteplici responsabilità su poche figure, la resistenza alla formalizzazione dei processi e una cultura tecnologica talvolta ancora legata a paradigmi tradizionali.

4.2 Evidenze empiriche sui benefici: il modello DORA

Il contributo più significativo alla comprensione empirica dei benefici di DevOps proviene dal programma di ricerca **DORA (DevOps Research and Assessment)**, condotto a partire dal 2014 da Nicole Forsgren, Jez Humble e Gene Kim [7] [8], e confluito nel volume

"Accelerate: The Science of Lean Software and DevOps" (2018). La ricerca, basata su migliaia di risposte raccolte a livello globale nel corso di diversi anni, ha stabilito correlazioni statisticamente significative tra la maturità delle pratiche DevOps e indicatori chiave di performance.

Il modello DORA identifica quattro metriche chiave, note come le "four key metrics", che sintetizzano le prestazioni di delivery del software [7]:

- **Deployment Frequency:** la frequenza con cui l'organizzazione rilascia in produzione. Le organizzazioni elite rilasciano su richiesta, anche più volte al giorno;
- **Lead Time for Changes:** il tempo dal commit alla produzione. Per le organizzazioni elite, questo tempo è inferiore a un'ora;
- **Change Failure Rate:** la percentuale di rilasci che causano fallimenti. Le organizzazioni elite mantengono questo valore tra lo 0% e il 15%;
- **Time to Restore Service:** il tempo di ripristino dopo un incidente. Per le organizzazioni elite, inferiore a un'ora.

Un risultato fondamentale della ricerca DORA è che queste quattro metriche non sono in conflitto tra loro: le organizzazioni più performanti eccellono simultaneamente in tutte e quattro le dimensioni. Questo sfata il mito, a lungo radicato nel settore, secondo cui velocità e stabilità sarebbero obiettivi in conflitto [7]. Al contrario, le pratiche che accelerano il rilascio (automazione, piccoli batch, feedback rapido) sono le stesse che migliorano la stabilità.

Forsgren, Humble e Kim hanno inoltre dimostrato che le capacità tecniche di continuous delivery, incluse l'automazione del deployment, la gestione della configurazione come codice e il monitoraggio proattivo, predicono in modo significativo sia le prestazioni di delivery sia le prestazioni organizzative (produttività, soddisfazione del team, riduzione del burnout). Il loro modello mostra che le capacità tecniche influenzano la cultura organizzativa e viceversa, in un ciclo virtuoso di miglioramento continuo.

Ulteriori evidenze provengono da studi di caso pubblicati da organizzazioni che hanno completato la transizione verso DevOps. Netflix, pioniera nell'adozione di microservizi e rilascio continuo, rilascia migliaia di volte al giorno su un'infrastruttura distribuita globalmente. Amazon effettua un deployment ogni 11,6 secondi in media, grazie a un'architettura fortemente decentralizzata basata su team autonomi ("two-pizza teams"). Etsy è passata da due rilasci alla settimana a cinquanta al giorno, documentando pubblicamente il percorso di trasformazione e le lezioni apprese [11]. Spotify ha introdotto il modello organizzativo a "squad, tribe, chapter, guild" specificamente per supportare l'autonomia dei team e la velocità di rilascio.

Questi esempi, pur provenendo da contesti enterprise di grandi dimensioni e con risorse significative, hanno avuto un impatto culturale profondo sull'intero settore. Essi hanno dimostrato che il rilascio continuo non è solo possibile ma è anche desiderabile, spostando la percezione del rilascio da evento eccezionale e rischioso a operazione di routine. Tuttavia, è importante contestualizzare questi risultati: le organizzazioni citate hanno investito anni e risorse ingenti nel raggiungere tali livelli di maturità. Per la maggior parte delle organizzazioni, l'obiettivo realistico non è rilasciare migliaia di volte al giorno, ma raggiungere un livello di automazione e affidabilità che elimini le principali fonti di rischio e inefficienza.

4.3 Sfide tecniche nell'adozione

Le sfide tecniche rappresentano il primo e più visibile ostacolo all'adozione di DevOps. La gestione di architetture legacy, spesso monolitiche e scarsamente documentate, rende complessa l'introduzione di pipeline automatizzate e di pratiche di rilascio continuo. La migrazione di applicazioni legacy verso architetture più adatte all'automazione è un processo lungo, costoso e rischioso che richiede competenze specifiche e una pianificazione attenta.

La complessità dell'ecosistema degli strumenti rappresenta un'altra sfida significativa. La necessità di integrare molteplici strumenti, ciascuno con la propria curva di apprendimento, può generare overhead di gestione e frammentazione delle competenze. Il fenomeno del "tool sprawl", ovvero la proliferazione incontrollata di strumenti, è particolarmente insidioso perché aumenta la complessità senza necessariamente migliorare i risultati.

La creazione e manutenzione di test automatici con copertura adeguata è una delle sfide più sottovalutate. La CI/CD è efficace solo nella misura in cui i test sono affidabili e rapidi. Test fragili (che falliscono in modo intermittente senza un difetto reale nel codice), test lenti (che rallentano la pipeline oltre i limiti di tolleranza del team) e test con copertura insufficiente (che lasciano passare difetti) compromettono l'intero processo. La costruzione di una suite di test di qualità è un investimento continuo che richiede disciplina e competenze specifiche.

La gestione degli ambienti e delle configurazioni rappresenta un ulteriore ambito di complessità. Garantire la coerenza tra ambienti di sviluppo, test, staging e produzione richiede investimenti in containerizzazione, IaC e gestione dei segreti. La divergenza tra ambienti è una delle cause più frequenti di errori in produzione, e la sua eliminazione richiede un approccio sistematico che non tutte le organizzazioni sono pronte a implementare.

4.4 Sfide organizzative e culturali

Le sfide organizzative e culturali sono spesso più difficili da affrontare di quelle tecniche, e la letteratura le identifica come il fattore più critico per il successo o il fallimento dell'adozione di DevOps. L'approccio DevOps richiede un cambiamento profondo nella cultura aziendale, che deve evolvere da un modello basato su silos funzionali a uno basato sulla collaborazione trasversale e sulla responsabilità condivisa.

Ron Westrum ha proposto una tipologia di culture organizzative, adottata dalla ricerca DORA [7] come framework di riferimento, che distingue tra culture **patologiche** (orientate al potere, caratterizzate da blame e segretezza), **burocratiche** (orientate alle regole, con responsabilità rigidamente compartimentalizzate) e **generative** (orientate alla performance, caratterizzate da alta collaborazione, condivisione delle responsabilità e orientamento alla mission). Le evidenze DORA mostrano che le organizzazioni con cultura generativa ottengono prestazioni DevOps significativamente superiori.

La resistenza al cambiamento è un fenomeno ampiamente documentato. I team abituati a lavorare in modo separato possono percepire l'adozione di DevOps come una minaccia alla propria autonomia o alle proprie competenze consolidate. La mancanza di supporto da parte del management è un ulteriore fattore di freno: senza un mandato chiaro e un investimento esplicito da parte della leadership, le iniziative DevOps tendono a rimanere circoscritte a singoli team.

La formazione e lo sviluppo delle competenze rappresentano un investimento necessario ma spesso sottovalutato. L'adozione di DevOps richiede competenze che spaziano dalla programmazione all'amministrazione di sistema, dalla conoscenza degli strumenti di automazione alla comprensione dei principi di sicurezza. La scarsità di figure professionali con competenze DevOps complete è un problema riconosciuto a livello globale.

Nelle piccole e medie imprese, queste sfide si manifestano in modo specifico. La limitatezza delle risorse rende difficile dedicare tempo e personale all'implementazione di nuovi processi. L'assenza di figure specializzate significa che le attività DevOps ricadono su sviluppatori che già svolgono molteplici ruoli, generando un sovraccarico che può compromettere sia la qualità dello sviluppo sia quella della gestione infrastrutturale. La pressione per la consegna di funzionalità lascia poco spazio all'investimento in automazione, creando un circolo vizioso in cui la mancanza di automazione genera inefficienze che a loro volta riducono il tempo disponibile per automatizzare.

4.5 Modelli di maturità DevOps

Per guidare le organizzazioni nel percorso di adozione, sono stati proposti diversi modelli di maturità DevOps [7] [8] che articolano il percorso in livelli progressivi, ciascuno caratterizzato da pratiche, strumenti e obiettivi specifici.

Un modello di maturità tipico può essere articolato in cinque livelli. Il livello iniziale è caratterizzato da processi manuali, assenza di versionamento strutturato e rilasci rari e rischiosi. Il livello gestito introduce il versionamento del codice con Git, build manuali ma documentate e test manuali sistematici. Il livello definito implementa pipeline CI automatizzate, test automatici di base e ambienti separati. Il livello quantitativamente gestito realizza pipeline CI/CD complete, deployment automatizzato, IaC e monitoraggio strutturato. Il livello ottimizzato raggiunge Continuous Deployment, rilascio progressivo, osservabilità avanzata e miglioramento continuo basato su metriche.

La tabella seguente sintetizza i cinque livelli del modello di maturità proposto, con le caratteristiche principali di ciascuno.

Livello	Denominazione	Caratteristiche principali
1	Iniziale	Processi manuali, rilasci rari e rischiosi, assenza di versionamento strutturato
2	Gestito	Git adottato, build documentate, test manuali sistematici
3	Definito	Pipeline CI automatizzata, test automatici di base, ambienti separati
4	Quantitativamente gestito	CI/CD completa, deploy automatizzato, IaC, monitoraggio
5	Ottimizzato	Continuous Deployment, rilascio progressivo, osservabilità avanzata, miglioramento continuo

Ogni organizzazione dovrebbe identificare il proprio livello attuale e definire un percorso di crescita realistico, con obiettivi intermedi raggiungibili e indicatori di progresso misurabili. Il passaggio da un livello al successivo non è necessariamente lineare né uniforme tra tutti gli aspetti del processo: un'organizzazione potrebbe trovarsi al livello 4 per il versionamento ma al livello 2 per il monitoraggio. Questa asimmetria è del tutto normale e anzi rappresenta un'indicazione preziosa per la prioritizzazione degli investimenti.

L'utilizzo delle quattro metriche DORA come indicatori di progresso è particolarmente raccomandato, in quanto esse forniscono una misura oggettiva e comparabile delle prestazioni di delivery [7] [8]. La misurazione regolare di queste metriche consente di rendere tangibili i miglioramenti ottenuti, di identificare le aree che richiedono attenzione e di mantenere il momentum del cambiamento organizzativo. Strumenti come DORA DevOps Quick Check, disponibile gratuitamente, consentono una prima valutazione rapida del proprio livello di maturità.

4.6 Strategie di adozione per organizzazioni con risorse limitate

Le organizzazioni con risorse limitate, in particolare le PMI e le startup in fase iniziale, necessitano di strategie di adozione pragmatiche che massimizzino il ritorno sull'investimento. La letteratura e la pratica professionale suggeriscono alcuni approcci particolarmente efficaci.

L'**approccio incrementale** prevede l'adozione graduale delle pratiche, partendo da quelle con il rapporto beneficio/costo più favorevole. L'introduzione del versionamento con Git e di una pipeline CI di base (build automatica e test unitari) rappresenta il punto di partenza con il maggiore impatto immediato [6] [20]. L'automazione del deployment può essere introdotta in un secondo momento, seguita dalla containerizzazione e dall'IaC.

L'**utilizzo di servizi cloud gestiti** consente di ridurre significativamente la complessità infrastrutturale. Piattaforme come GitHub Actions e GitLab CI/CD offrono funzionalità CI/CD senza richiedere la gestione di server dedicati. I tier gratuiti di queste piattaforme sono spesso sufficienti per team di piccole dimensioni, rendendo possibile l'adozione iniziale a costo zero.

Un'alternativa emergente è il modello di **DevOps as a Service**, in cui le competenze DevOps vengono esternalizzate a fornitori specializzati. Questo modello consente alle organizzazioni di accedere a competenze specializzate senza svilupparle internamente, riducendo il time-to-value e liberando il team di sviluppo per le attività core. Le modalità di erogazione possono variare dal supporto continuo alla consulenza su richiesta, fino all'implementazione di progetti specifici come la costruzione di pipeline o la migrazione a container.

La formazione interna, anche a livelli basilari, rappresenta un investimento con ritorni elevati. Workshop pratici, pair programming su attività di automazione, la designazione di un "DevOps champion" all'interno del team e la partecipazione a comunità di pratica sono approcci a basso costo che possono accelerare significativamente il percorso di adozione.

4.7 Tendenze emergenti

Platform Engineering si sta affermando come evoluzione naturale di DevOps. Il suo obiettivo è la creazione di piattaforme interne di sviluppo (Internal Developer Platforms, IDP) che astraggono la complessità infrastrutturale e offrono ai team di sviluppo un'interfaccia semplificata e self-service per il deploy e la gestione delle applicazioni. Platform Engineering riconosce che non tutti gli sviluppatori possono o vogliono acquisire competenze infrastrutturali approfondite, e propone di concentrare tali competenze in team dedicati che costruiscono strumenti per gli altri.

GitOps, paradigma proposto da Weaveworks, utilizza Git come unica fonte di verità per la configurazione sia applicativa sia infrastrutturale. In un approccio GitOps, ogni modifica all'infrastruttura o alla configurazione avviene attraverso un commit nel repository, e un operatore automatizzato (come ArgoCD o Flux) si occupa di riconciliare lo stato effettivo dell'infrastruttura con lo stato desiderato descritto nel repository. Questo approccio offre tracciabilità completa, rollback semplificato e un audit trail nativo.

L'integrazione dell'intelligenza artificiale nelle pipeline CI/CD rappresenta un'ulteriore frontiera. L'AI può essere impiegata per l'analisi predittiva dei fallimenti, l'ottimizzazione automatica delle suite di test (identificando i test più rilevanti per ciascuna modifica), la generazione automatica di test, la rilevazione di anomalie post-deploy e l'assistenza alla risoluzione degli incidenti. Sebbene queste applicazioni siano ancora in fase di maturazione, il loro potenziale è significativo e promette di ridurre ulteriormente la barriera all'ingresso per l'adozione di pratiche DevOps avanzate.

Un tema trasversale a tutte queste tendenze è la crescente attenzione alla developer experience (DevEx): la qualità dell'esperienza dello sviluppatore nell'interazione con gli strumenti, i processi e l'infrastruttura. La ricerca ha dimostrato che una migliore DevEx correla con maggiore produttività, minore turnover e maggiore qualità del software. Investire nella DevEx significa ridurre le frizioni, automatizzare le attività ripetitive e consentire agli sviluppatori di concentrarsi sulle attività a maggiore valore aggiunto.

Capitolo 5 – Conclusioni e Sviluppi Futuri

5.1 Sintesi del lavoro svolto

Il presente lavoro di tesi ha analizzato in modo approfondito i principi, gli strumenti e le sfide che caratterizzano l'approccio DevOps e le pratiche di Continuous Integration e Continuous Delivery, con l'obiettivo di fornire un quadro teorico completo e aggiornato del panorama attuale.

Attraverso l'analisi della letteratura scientifica e professionale, sono stati esaminati i fondamenti teorici dell'ingegneria del software e la loro evoluzione verso i paradigmi Agile e DevOps, l'ecosistema degli strumenti disponibili per l'automazione del ciclo di rilascio, le sfide tecniche, organizzative e culturali dell'adozione e le evidenze empiriche disponibili sui benefici delle pratiche CI/CD.

L'approccio adottato ha consentito di integrare la prospettiva tecnica, relativa agli strumenti e alle architetture, con quella organizzativa, relativa alla cultura, ai processi e alle strategie di adozione, offrendo una visione multidimensionale del fenomeno.

5.2 Principali evidenze

Dall'analisi condotta emergono alcune evidenze chiave. In primo luogo, DevOps rappresenta un cambio di paradigma che va ben oltre l'adozione di strumenti specifici. La dimensione culturale e organizzativa è almeno tanto importante quanto quella tecnica, e il successo dell'adozione dipende dalla capacità dell'organizzazione di evolvere verso modelli collaborativi e orientati al miglioramento continuo. La tipologia di Westrum e le evidenze DORA confermano che la cultura generativa è un prerequisito per prestazioni DevOps elevate [16] [7].

In secondo luogo, l'ecosistema degli strumenti DevOps ha raggiunto un livello di maturità che lo rende accessibile a organizzazioni di qualsiasi dimensione. Le piattaforme cloud e i servizi gestiti hanno significativamente abbassato la barriera d'ingresso, rendendo possibile l'implementazione di pipeline CI/CD anche in contesti con risorse limitate. La disponibilità di tier gratuiti e di soluzioni SaaS elimina la necessità di investimenti iniziali in infrastruttura, rendendo l'adozione una scelta praticabile anche per le microimprese.

In terzo luogo, le evidenze empiriche disponibili, in particolare quelle del rapporto DORA e dello studio "Accelerate" [7], confermano in modo robusto che le organizzazioni con pratiche DevOps mature ottengono prestazioni significativamente superiori sia in termini

tecnologici (frequenza di rilascio, stabilità, tempi di ripristino) sia organizzativi (produttività, soddisfazione, riduzione del burnout). Un risultato particolarmente significativo è che velocità e stabilità non sono in conflitto, ma si rafforzano reciprocamente [7].

Infine, l'adozione di DevOps è un percorso graduale che richiede una strategia esplicita, obiettivi intermedi misurabili e un investimento costante. I modelli di maturità forniscono un framework utile per orientare il percorso e misurare i progressi. Le tendenze emergenti come Platform Engineering, GitOps e l'integrazione dell'AI promettono di rendere l'automazione sempre più accessibile e sofisticata.

5.3 Implicazioni per le organizzazioni

I risultati dell'analisi hanno implicazioni rilevanti per le organizzazioni che intendono avviare o accelerare il proprio percorso di adozione DevOps.

Per le grandi imprese, la sfida principale risiede nella gestione della trasformazione culturale e nella modernizzazione delle architetture legacy. L'approccio raccomandato è quello di iniziare con progetti pilota, dimostrare il valore attraverso quick wins misurabili e poi estendere gradualmente le pratiche all'intera organizzazione. La creazione di team platform dedicati può accelerare l'adozione riducendo la complessità per i team di sviluppo.

Per le piccole e medie imprese, la strategia più efficace è l'adozione incrementale, partendo dal versionamento del codice e dalla CI di base, per poi evolvere verso l'automazione completa del deployment. L'utilizzo di servizi cloud gestiti e, ove necessario, il ricorso a modelli di DevOps as a Service possono compensare la limitatezza delle risorse interne. La designazione di un DevOps champion all'interno del team, anche senza un'assunzione dedicata, può fornire il punto di riferimento necessario per avviare e mantenere il percorso di adozione.

Indipendentemente dalla dimensione, tutte le organizzazioni beneficiano di un investimento nella formazione delle competenze DevOps, nella definizione di metriche chiare per misurare il progresso (le quattro metriche DORA rappresentano un riferimento consolidato) e nella creazione di una cultura orientata alla collaborazione, alla trasparenza e al miglioramento continuo.

5.4 Contributo del lavoro

Il contributo principale di questo lavoro risiede nella sistematizzazione e nell'aggiornamento del quadro teorico relativo a DevOps e CI/CD, con un'attenzione particolare alle implicazioni pratiche per le organizzazioni di diversa dimensione. La trattazione integrata

degli aspetti tecnici (strumenti, pipeline, architetture), organizzativi (cultura, processi, ruoli) e strategici (modelli di adozione, maturità, servizi esterni) offre una risorsa utile sia per la comprensione accademica del fenomeno sia per l'orientamento delle decisioni operative.

L'analisi delle sfide specifiche delle organizzazioni con risorse limitate e la discussione delle strategie di adozione pragmatiche colmano un gap nella letteratura, che tende a focalizzarsi sulle best practice delle grandi organizzazioni tecnologiche senza considerare le condizioni specifiche dei contesti più vincolati.

5.5 Limiti dello studio e sviluppi futuri

Il principale limite del presente lavoro è la sua natura prevalentemente teorica e analitica, basata sulla revisione della letteratura esistente. Sebbene l'analisi si fondi su fonti autorevoli e aggiornate, l'assenza di un'indagine sul campo limita la possibilità di verificare direttamente le ipotesi formulate e di adattare le conclusioni a contesti specifici. Un ulteriore limite riguarda la rapidità dell'evoluzione dell'ecosistema DevOps, che rende inevitabilmente datate alcune informazioni relative agli strumenti nel giro di pochi mesi.

A partire dal lavoro svolto, è possibile identificare diversi sviluppi futuri. In primo luogo, la conduzione di un'indagine empirica consentirebbe di verificare le ipotesi formulate e di arricchire il quadro teorico con evidenze raccolte direttamente dai professionisti del settore. A tal fine, è stato progettato un questionario strutturato, composto da trenta domande organizzate in sei sezioni tematiche, che copre il profilo dei rispondenti, la gestione del versionamento e dei rilasci, i problemi ricorrenti, il valore percepito dell'automazione, il livello di maturità CI/CD e l'interesse verso servizi esterni.

Il questionario, realizzato tramite Google Forms, prevede l'utilizzo di una scala Likert a 5 punti per la misurazione delle percezioni e la costruzione di un Indice di Maturità CI/CD, ottenuto aggregando le risposte a un sottoinsieme di domande relative a competenza nell'uso di Git, automazione del rilascio, frequenza di problemi e autovalutazione della maturità. La somministrazione e l'analisi dei risultati di tale questionario, destinato a professionisti operanti in contesti aziendali eterogenei (startup, PMI, grandi aziende), costituiscono il naturale proseguimento di questo lavoro. L'eterogeneità del campione consentirà di confrontare il livello di adozione tra organizzazioni di diversa dimensione, verificando empiricamente le differenze ipotizzate sulla base della letteratura.

Inoltre, l'approfondimento delle tendenze emergenti come Platform Engineering, GitOps e l'integrazione dell'intelligenza artificiale nelle pipeline CI/CD rappresenta un'area di ricerca promettente, così come lo sviluppo di un modello di maturità DevOps specifico per le PMI italiane, che tenga conto dei vincoli di risorse e delle specificità organizzative di queste realtà.

5.6 Considerazioni finali

In conclusione, DevOps rappresenta non solo un insieme di pratiche tecniche, ma un vero e proprio cambio di paradigma nella gestione del ciclo di vita del software. La sua adozione, pur non priva di sfide, offre benefici dimostrabili e significativi in termini di qualità, efficienza e competitività.

Le organizzazioni di ogni dimensione hanno l'opportunità di trarre vantaggio dall'adozione di DevOps, a patto di affrontare il cambiamento con un approccio consapevole e graduale, che integri la dimensione tecnica con quella culturale e organizzativa. Il miglioramento continuo dei processi di sviluppo e rilascio del software non è un traguardo da raggiungere una volta per tutte, ma un percorso permanente che richiede investimento, dedizione e apertura al cambiamento.

L'auspicio è che questo studio possa offrire un contributo utile per comprendere lo stato attuale delle pratiche di rilascio e per orientare le scelte di chi intende intraprendere il percorso verso una gestione più matura, automatizzata e affidabile del ciclo di vita del software.

Bibliografia

[1] Anderson, D.J. (2010). Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press.

[2] Bass, L., Weber, I. e Zhu, L. (2015). DevOps: A Software Architect's Perspective. Addison-Wesley Professional.

[3] Beck, K. (1999). Extreme Programming Explained: Embrace Change. Addison-Wesley.

[4] Boehm, B.W. (1988). "A Spiral Model of Software Development and Enhancement". Computer, 21(5), pp. 61–72.

[5] Burns, B. et al. (2016). "Borg, Omega, and Kubernetes". ACM Queue, 14(1), pp. 70–93.

[6] Duvall, P.M., Matyas, S. e Glover, A. (2007). Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley.

[7] Forsgren, N., Humble, J. e Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps. IT Revolution Press.

[8] Forsgren, N., Tremblay, M.C., VanderMeer, D. e Humble, J. (2017). "DORA platform: DevOps assessment and benchmarking". In International Conference on Design Science Research in Information System and Technology, pp. 436–440. Cham: Springer International Publishing.

[9] Humble, J. e Farley, D. (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley.

[10] Kim, G., Behr, K. e Spafford, G. (2013). The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win. IT Revolution Press.

[11] Kim, G., Debois, P., Willis, J. e Humble, J. (2016). The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press.

[12] Morris, K. (2016). Infrastructure as Code: Managing Servers in the Cloud. O'Reilly Media.

[13] Pressman, R.S. e Maxim, B.R. (2019). Software Engineering: A Practitioner's Approach. 9th Edition. McGraw-Hill Education.

[14] Royce, W.W. (1970). "Managing the Development of Large Software Systems". Proceedings of IEEE WESCON, pp. 1–9.

[15] Sommerville, I. (2015). Software Engineering. 10th Edition. Pearson Education.

[16] Westrum, R. (2004). "A Typology of Organisational Cultures". BMJ Quality & Safety, 13(suppl 2), pp. ii22–ii27.

Sitografia

[17] Beck, K. et al. (2001). Manifesto for Agile Software Development. Disponibile su: https://agilemanifesto.org

[18] Debois, P. (2009). DevOpsDays Ghent 2009. Disponibile su: https://devopsdays.org

[19] Driessen, V. (2010). A Successful Git Branching Model. Disponibile su: https://nvie.com/posts/a-successful-git-branching-model

[20] Fowler, M. (2006). Continuous Integration. Disponibile su: https://martinfowler.com/articles/continuousIntegration.html

[21] Puppet Labs e DORA (2023). State of DevOps Report. Disponibile su: https://cloud.google.com/devops

[22] Schwaber, K. e Sutherland, J. (2020). The Scrum Guide. Disponibile su: https://scrumguides.org