Alma Mater Studiorum · Università di Bologna ·
Campus di cesena

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea Magistrale in Ingegneria e scienze informatiche

DA SOFTWARE MONOLITICO A DEVOPS E MICROSERVIZI: UN CASO DI STUDIO INDUSTRIALE

Tesi di Laurea Magistrale in
Laboratorio di Sistemi Software LM

Relatore:
Danilo Pianini

Presentata da:
Alessandro Neri

Sessione Terza
Anno Accademico 2019/2020

Sommario

Uno dei fattori che determina il successo di un'azienda informatica consiste nella capacità di produrre codice di qualità, che sappia catalizzare i desideri della propria clientela, in un ambiente di business dinamico e altamente competitivo. Per questo motivo, rimanere al passo con le tecnologie e i progressi dell'ingegneria del software, non è più una scelta ma bensì una necessità.

In tale contesto spiccano approcci architetturali di ingegneria del software che favoriscono la componibilità, la scalabilità e l'isolamento. Le architetture a microservizi sono un tipo di architettura che consiste nello scomporre le applicazioni in servizi di piccole dimensioni, (semi) autonomi, isolati (solitamente containerizzati) e che adempiono ad una funzione ben precisa. La tendenza verso architetture a microservizi è sostenuta e amplificata dal crescente interesse verso la filosofia DevOps. DevOps sfida il modo tradizionale di sviluppare i sistemi, introducendo princìpi che promuovono l'abbattimento delle barriere tra team organizzati verticalmente all'interno di un'azienda. Da questi princìpi nascono pratiche mirate a massimizzare la qualità del prodotto e al contempo minimizzare la durata che intercorre tra l'applicazione di un cambiamento nella base di codice e la sua effettiva applicazione nell'ambiente di produzione.

In questa tesi si è documentato il processo che ha portato un gruppo ristretto di sviluppatori di una grande azienda del territorio a riprogettare un prodotto software monolitico e il relativo processo di sviluppo attraverso la trasformazione architetturale a microservizi e l'adozione della filosofia DevOps. L'obiettivo è quello di dimostrare, tramite comparazione di metriche e indicatori di prestazione, i benefici ottenuti dall'applicazione combinata dei concetti derivanti dai due mondi. Il problema si considera sotto diverse prospettive di coordinamento: quello dei processi di sviluppo, dei servizi applicativi (livello architetturale) e dei servizi operativi (pipeline DevOps). Dapprima si è condotta una ricerca sullo stato dell'arte relativo a princìpi, pratiche e strumenti DevOps. Successivamente si riporta l'esperienza di implementazione del rinnovo architetturale, tecnologico e dei processi dell'applicativo software industriale. Durante l'esposizione sarà posta particolare attenzione alle pratiche DevOps che hanno consentito la trasformazione e il loro effetto sulla qualità del software e del processo rispetto al sistema originale.

Indice

Capitolo 1

DevOps e stato dell'arte

1.1 Introduzione

Per tantissimi anni le aziende nel settore IT hanno mantenuto un approccio strutturato in cui ogni fase del il ciclo di vita del software (es: sviluppo, test, controllo della qualità, rilascio, monitoraggio, eccetera) è suddivisa tra diversi team strutturati verticalmente e comunemente chiamati Silos. Si opera con obiettivi non sempre compatibili e con definizioni diverse di "lavoro completato" (per alcuni significa codice sviluppato, per altri codice testato oppure pacchetto rilasciato). Nella maggior parte dei casi l'applicazione passa da un team all'altro tramite una gestione manuale che prevede il minimo sforzo di comunicazione [15].

Da una parte gli sviluppatori (Dev) devono sviluppare codice con un ritmo che segua le continue richieste del cliente mentre dall'altra vi sono i team operativi (Ops) con il compito di gestire l'infrastruttura, rilasci, configurazioni e controllo di qualità (QA). Una separazione così netta tra entità tra loro dipendenti ha portato a prodotti con tempi di produzione sempre maggiore e con una qualità non sempre ottimale.

Anche se in letteratura DevOps risponde a diverse definizioni, queste concordano sugli obiettivi e sui mezzi utilizzati per raggiungerli: DevOps ha lo scopo di ridurre il tempo tra l'applicazione di un cambiamento nella base di codice e la sua effettiva reificazione nell'ambiente di produzione, senza alcuna perdita (anzi, con un incremento) di qualità; questo è ottenuto attraverso quattro princìpi: collaborazione, automazione, misurazione e monitoraggio. In questo capitolo esporremo una tassonomia di significati e concetti relativi all'ecosistema DevOps che sarà la base su cui si fonderà il lavoro di tutto il documento.

1.2 Visione e principi

Da uno studio svolto sulla base di numerose pubblicazioni [26], i diversi significati che in letteratura si attribuiscono al movimento DevOps condividono 3 visioni principali:

- visione **generica**: "una metodologia di sviluppo software che combina i meccanismi per il controllo di qualità alle operazioni del settore IT che compongono le pratiche di ingegneria del software". Altre definizioni trattano di approcci e non di metodi.

- visione **tecnica**: DevOps mira ad automatizzare l'intero processo di rilascio, dai sorgenti all'ambiente di produzione. Si richiamano spesso le nozioni di "DevOps toolchain" e le pratiche di "feedback rapido, piccoli set di modifiche e rilasci indipendenti". Allo stesso modo, per [27] DevOps significa "un gruppo di princìpi riguardo le metodologie e la velocità di rilascio del software [...], il testing continuo in ambienti simil produzione, branch sempre pronte per essere rilasciabili, feedback continuo, reattività ai cambiamenti, gruppi di lavoro che collaborano per il raggiungimento di un obiettivo invece che di un compito".
- visione **olistica**: In [16], Jez Humble dice "Devops riguarda la capacità di allineare gli obiettivi di tutte le persone coinvolte nella produzione del software". Obiettivo raggiungibile perseguendo: "cultura, automazione, misura, e condivisione"

L'interpretazione che verrà adottata da qui in avanti sarà un ibrido tra le varie visioni. Parlare di DevOps, vuol dire inquadrare un tipo di cultura, una cultura che promuove la collaborazione e l'integrazione tra i team di sviluppo e quello operativo con lo scopo di produrre risultati in maniera continua ed efficace.

Abbracciare una cultura significa aderire ad una serie di **princìpi**, che ispirano **pratiche** che richiedono **strumenti** di lavoro[1]:

- Collaborazione: Dev e Ops lavorano insieme
- Gruppi autonomi: Un gruppo ha responsabilità su tutto il ciclo di vita del prodotto. Nessun passaggio di artefatti tra gruppi diversi (es: da sviluppatori a gruppo operativo). Il processo decisionale risulterà più rapido.
- Riproducibilità
- Robustezza
- Automazione: limitare, ove possibile, processi che necessitano di intervento umano. Riduzione di tempi di overhead e possibili errori.
- Continuous Improvement: i cambiamenti avvengono di continuo. DevOps consiste nell'essere pronti a recepirli, adattandosi rapidamente a nuovi processi, tecnologie e buone pratiche.
- Focus sul processo, e non sul prodotto: promuovere piccole modifiche incrementali. Automatizzare il più possibile e scegliere lo strumento che meglio si adatta al lavoro assegnato.
- Misura tutto: non ci può essere automazione senza la capacità di analizzare lo stato dei processi che compongono il sistema.

Questo princìpi vengono spesso riassunti e referenziati tramite il *modello CAMS*, inventanto da Damen Edwards e John Willis. CAMS è l'acronimo di **C**ultura, **A**utomazione, **M**isurazione e Condivisione (**S**haring)

1.3 Benefici

La rottura delle barriere tra Dev e Ops, unita ai princìpi che fanno da base a questa cultura, si traducono in una serie di vantaggi per tutte le organizzazioni che ne aderiscono.

[1] `https://github.com/DanySK/Course-Laboratory-of-Software-Systems`

1.3.1 Collaborazione

DevOps si basa largamente sull'unione degli sviluppatori con il team operativo. Una cultura che promuove la collaborazione crea continuamente opportunità di scambio tra persone che possiedono diverse conoscenze, abilità e competenze.

Strutturare team in modo che sappiano operare su tutta la catena di produzione e rilascio del software, aumenta sicuramente la qualità su tutto il processo di sviluppo. Lavorando come un entità unica, ora i due team possono sperimentare, ricercare e innovare in modo più efficace.

1.3.2 Accelerazione dell'innovazione

Integrando il team di Dev e Ops le applicazioni possono essere sviluppate e distribuite molto più rapidamente. Questa rapidità è fondamentale poiché il successo aziendale dipende in gran parte dalla capacità di un'organizzazione di innovare più velocemente della concorrenza. Con set di modifiche più piccole, i problemi tenderanno ad essere meno complessi. Gli ingegneri DevOps possono sfruttare i dati sulle prestazioni in tempo reale per valutare rapidamente l'impatto delle modifiche sulle applicazioni. Le correzioni al software sono più veloci perché gli sviluppatori devono testare solo le ultime modifiche al codice per verificare la presenza di errori.

1.3.3 Gestione del rischio

I cicli di sviluppo più brevi associati a un forte approccio DevOps promuovono rilasci di codice più frequenti. Con queste implementazioni più modulari, bug nel software, problemi di configurazione o nell'infrastruttura, emergono nelle prime fasi del ciclo di rilascio. DevOps tiene inoltre impegnati tutti i membri del team per tutto il ciclo di vita di una funzionalità o applicazione, con conseguente codice di qualità superiore. Sono necessarie meno correzioni negli ambienti di test e produzione perché gli sviluppatori ricevono feedback continuo già durante la scrittura del codice.

Anche quando un bug raggiunge gli ambienti di produzione, grazie alla pratica di rilasci continui, sarà facile determinare le modifiche che hanno causato il problema. In generale si ottiene un'ottimizzazione dell'MTTR (Mean Time to Recovery[2])

Secondo un recente rapporto sullo stato di DevOps, le organizzazioni che adottano una cultura DevOps riscontrano circa 60 volte meno errori rispetto a quelle che non implementano un approccio DevOps [12].

1.3.4 Aumento dell'efficienza

Sulla base del principio di automazione nascono strumenti e piattaforme, mirate alle varie fasi del ciclo di produzione del software. Oltre a rendere i rilasci più prevedibili (riducendo la variabilità data dall'errore umano), liberano il personale IT da noiose e ripetitive attività. Con la compilazione e test automatici, gli sviluppatori lasciano il compito di

[2]Mean time to recovery (MTTR) è il tempo che mediamente impiega un dispositivo per tornare operativo dopo un guasto. Esempi vanno dal semplice riavvio (MTTR di pochi secondi) a complete sostituzione del dispositivo.

integrazione ad un processo terzo, guadagnando tempo prezioso per la produzione di software.

Le piattaforme offrono ulteriori opportunità per migliorare l'efficienza:

- *Self-Service e infrastruttura scalabile*, come le soluzioni basate su cloud, aiutano ad accelerare i processi di test e distribuzione facilitando l'accesso alle risorse hardware.
- Gli strumenti di *Build Automation* (Automazione dello sviluppo) aiutano ad abbreviare i cicli di sviluppo e ad accelerare la consegna del prodotto.
- I flussi di *Continuos Delivery* aiutano a produrre rilasci software più rapidi e frequenti.

1.3.5 Migliora la soddisfazione sul lavoro

Piuttosto che una cultura basata su regole o sul potere, DevOps promuove un ambiente aziendale basato sulle prestazioni. Ciò riduce gli ostacoli burocratici e favorisce la condivisione dei rischi. Il risultato è una forza lavoro più soddisfatta e produttiva[3], che aiuta a migliorare la qualità dell'organizzazione. Gli sviluppatori e gli ingegneri operativi generalmente preferiscono un ambiente DevOps perché possono lavorare in modo più efficiente e indossando più di un cappello. Inoltre acquisiscono una migliore comprensione del loro ruolo all'interno dell'ampio ambito dell'IT e all'interno dell'azienda nel suo complesso [13].

1.4 Sfide

Adottare DevOps significa essere disposti ad affrontare diverse tipologie di sfide. Come abbiamo visto, i princìpi toccano sia aree relative alla comunicazione e organizzazione dei gruppi, sia aspetti più tecnici. Per di più i due team principali, Dev e Ops, un tempo suddivisi e abituati a ragionare secondo prospettive diverse, ora operano a stretto contatto. Ad esempio potenziali obiettivi del team operativo saranno quelli di gestire la stabilità e affidabilità del sistema, mentre il team di sviluppo ricercherà innovazione negli strumenti e tecnologie che utilizzano tutti i giorni. Il problema si pone nel momento in cui si vogliono far dialogare questi mondi.

Dal punto di vista tecnico, molte sfumature dei concetti DevOps, si riconducono alla ricerca di automazione. In alcuni scenari il rilascio di un'applicazione viene fatto numerose volte durante la stessa giornata. Questo significa dover prevedere meccanismi di ripristino che agiscano in automatico quando certi KPI[4] di produzione non sono soddisfatti. E quando fallisce anche il ripristino automatico come si procede? Di chi è la responsabilità [19]?

Inoltre, in molte grandi organizzazioni, nonostante siano già in uso pipeline automatiche che gestiscono il ciclo di vita del software, si mantengono pratiche manuali legate al radicato utilizzo di metodologie non agili.

Il miglior modo per trasformare un processo ben definito negli anni, in uno più efficiente, consiste nel saper mappare le attività del processo e analizzare in modo oggettivo cosa funziona e cosa non funziona; su tale analisi sarà possibile mettere in evidenza problemi di

[3] `https://archive.is/Mg6C4`
[4] Indicatori chiave di prestazione

processo legati a sprechi di risorse, mancanza di comunicazione e poca automatizzazione delle attività.

1.5 Pratiche e strumenti

Nel capitolo precedente si è data una definizione dei princìpi e concetti che costituiscono la cultura DevOps. Saranno ora analizzate quelle che ad oggi sono le principali pratiche nate sulla base di tali principi ed i relativi strumenti adottati dal settore IT.

1.5.1 Organizzazione del flusso di lavoro

Parlare di DevOps non significa richiamare l'uso di uno specifico processo o framework. Non è una soluzione specifica nata per risolvere un problema di ingegneria del software (vedi Scrum). DevOps, come cultura, promuove un tipo di collaborazione trasversale, che avvicina e unisce team un tempo visti come verticali isolati, riprogettandone la struttura organizzativa [23]. Tuttavia questo non significa che un'organizzazione deve liberarsi di metodologie e sistemi preesistenti, perché l'integrazione con i principi di DevOps rimane sempre possibile.

Ad esempio, *Disciplined Agile Delivery* è un processo aziendale che include in livello di DevOps nel proprio modello, chiamato Disciplined DevOps. Lo stesso vale per la metodologia Scrum unita a componenti della DevOps: *Continuous Scrum*. Uno dei modelli più adottati è quello che si basa sulla metodologia di sviluppo Agile ed in particolare sul ciclo di vita del software da essa promosso.

Adottare una metodologia significa definire **cosa**, **come** e **dove** si dovranno applicare e mantenere le diverse pratiche promosse dalla cultura DevOps. Il ruolo dei partecipanti dovrà essere chiaro e ben definito.

Strumenti:

- Strumenti di versionamento del codice (es: Git, Svn)
  1. politica di branching: Git-flow, Feature Branch, ecc.
  2. semantic commit, commento libero
  3. politica di versioning: semantic versioning, basato su date, nome in codice, ecc.
- Strumenti di gestione dei processi di sviluppo
  1. Monitoraggio dell'avanzamento
  2. Gestione delle problematiche
  3. Gestione nuove proposte
  4. Pianificazione delle Milestone
  5. Servizio di Service Desk
  6. Monitoraggio degli ambienti
  7. Sistemi di comunicazione
  8. Sistemi di compilazione

Sono sempre più popolari, piattaforme *all-in-one*, ovvero piattaforme che uniscono e integrano tutti questi strumenti: ad esempio Github, Gitlab, Bitbucket.

1.5.2 Continuous Integration

Anche conosciuta come CI, è una pratica che consiste nella periodica integrazione del lavoro degli sviluppatori nella base di codice principale. Come si vede in fig. 1.1, nasce dal principio che integrare spesso tante piccole modifiche è meno rischioso che integrare grandi modifiche con meno frequenza. Il vantaggio principale ottenuto da questa pratica è quello di non avere mai rami di sviluppo che **divergano** incontrollabilmente da quello principale.

Il successo di questa pratica spesso è determinato dalla sensibilità del singolo sviluppatore che, seppur incoraggiato ad integrare anche più volte al giorno, a lui rimane la scelta finale di quando effettivamente farlo. Se applicata correttamente, la difficoltà relativa all'integrazione di basi di codice si riduce in quanto, con l'introduzione di piattaforme di CI, è possibile sapere per ogni commit il risultato della sua integrazione in un determinato ramo, facilitando l'intercettazione di problematiche già nelle prime fasi di lavoro.

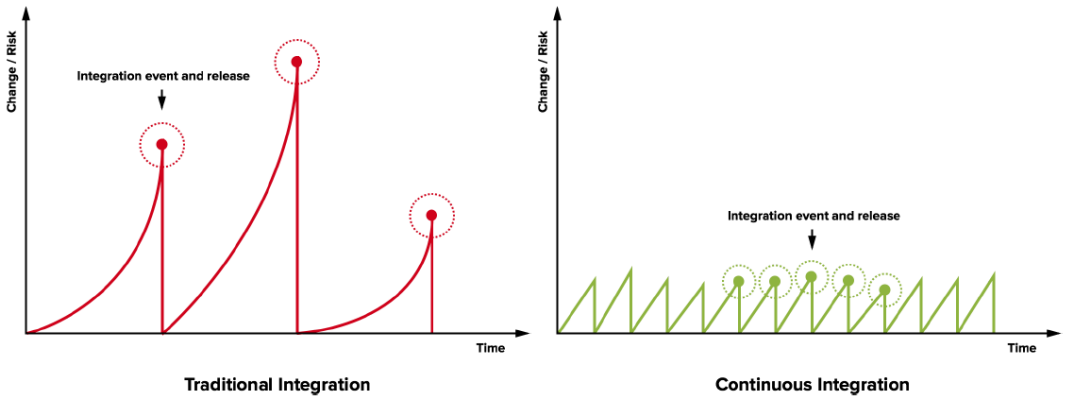

Figura 1.1: Confronto del rischio con e senza CI

Il successo nell'adozione della CI si basa su accorgimenti che influenzano il lavoro di tutti i giorni: il codice è gestito da una piattaforma di versionamento (non scontato, soprattutto per le piccole organizzazioni) [22].

Quando uno sviluppatore applica una modifica alla base di codice, un sistema automatico (piattaforma CI) deve recepire la modifica, scaricarsi il codice aggiornato, e lanciare una serie di attività che determinano se la modifica introduce cambiamenti che non invalidano la stabilità del ramo. Il sistema va considerato come un giudice imparziale, e va progettato per risolvere la sindrome del "funziona sulla mia macchina". E' fondamentale che i processi applicati dal sistema di CI siano totalmente automatici e che non richiedano l'intervento del programmatore a garanzia della sua produttività.

Di seguito una raccolta delle principali sfide a cui rispondere nel momento in cui si adotta la pratica di integrazione continua:

- **Build Stabili**: con l'aumento della frequenza di integrazione, mantenere stabile il ramo principale diventa di vitale importanza. Compromettere questa stabilità significa ridurre la confidenza con la quale lo sviluppatore applica e integra le proprio modifiche, proprio perché non è possibile risalire al momento storico in cui è stato introdotto il problema. Una pipeline di integrazione risulta in errore (red) per diversi

motivi: errori di compilazione, test falliti, quality gate falliti durante analisi statica. Una green build è uno dei requisiti per considerare rilasciabili in produzione le nuove modifiche.

- **Testing**: la buona riuscita delle pratica di CI consiste nell'avere un adeguato sistema automatico in grado di validare il buono stato del sistema. I test, strutturati su vari livelli, sono lo strumento più potente in mano al server di CI. Tuttavia per essere efficaci, occorre che i test siano scritti e mantenuti in modo che coprano la maggior parte del codice. La metrica che si adotta in questi casi è chiamata Test Coverage. Con l'evolversi del codice sorgente e delle funzionalità, anche i test necessitano di continua manutenzione. Altrimenti saranno anch'essi principale fonte di errore durante le compilazioni.
- **Coordinamento dell'integrazione**: una conseguenza delle integrazioni più frequenti quando si adottano pratiche di CI è che la suddivisione del lavoro diviene più granulare. Ciò significa che il lavoro che un tempo sarebbe stato sviluppato in un unica tranche, ora potrebbe essere necessario suddividerlo in più integrazioni. Inoltre, la suddivisione implica che lo sviluppo potrebbe essere effettuato da diversi sviluppatori invece di uno solo. Questo pone attenzione su diverse problematiche. La prima è quella di riuscire a tenere traccia di tutte le integrazioni che vengono riportare sul ramo di sviluppo, e la loro relazione con le attività di lavoro. La seconda è una problematica di responsabilità: chi interviene quando si presenta una situazione di integrazione fallita? Per avere maggior controllo rispetto a situazioni di questo tipo, alcuni team adottano un meccanismo a pull request (anche conosciuto come merge request), in cui lo sviluppatore non è in grado di riversare in autonomia le proprie modifiche, ma deve sottoporre una richiesta. Nel meccanismo a pull request, quando tutti i controlli automatici terminano con successo, una o più persone con l'incaricato di validare la bontà dei commit (**code review**) decideranno se le modifiche proposte verranno accettate o meno.

1.5.3 Continuous Testing

I test vengono eseguiti frequentemente durante lo sviluppo per garantire l'affidabilità del software rilevando bug ed errori di regressione. Tuttavia, fermarsi spesso anche per eseguire il test fa perdere tempo rallentando i progressi dello sviluppo.

Con questa pratica si vuole dapprima progettare i diversi livelli di test (unit, acceptance, integration, smoke test, end to end) tramite un build automator che possa predisporre le condizioni di esecuzione. Successivamente si automatizza tale esecuzione in modo tale che possa intercettare problematiche durante tutta la fase di sviluppo del software: durante lo sviluppo (ambiente locale), durante il rilascio (piattaforma di rilascio) e post rilascio (ambienti di staging).

È buona pratica organizzare il flusso di sviluppo in modo che sia necessario rispettare una percentuale minima di copertura del codice (*code coverage*) quando si introduce una nuova funzionalità. In generale risolvere un problema significa introdurre test di unitari di regressione.

1.5.4 Continuous Inspection

La filosofia su cui si base è quella di individuare il prima possibile ed in modo completamente automatico, tutte quelle problematiche che sono ricavabili staticamente senza necessità di avere il sistema in esecuzione. Molti degli strumenti associati a questa pratica forniscono la possibilità di impostare dei **quality gate**: ovvero delle soglie che, se non rispettate, interrompono il processo di compilazione.

Generalmente le ricerche avvengono su codice, sistemi e artefatti al fine di individuare:

- errori nel codice.
- code smell, letteralmente "puzza del codice", viene usata per indicare una serie di caratteristiche che il codice sorgente può avere e che sono generalmente riconosciute come probabili indicazioni di un difetto di programmazione.
- vulnerabilità di sicurezza.
- mancato adempimento a definizioni di regole di stile.
- stato delle dipendenze (problemi di incompatibilità, licenze, aggiornamenti).

1.5.5 Continuous Delivery

Anche conosciuta come CDE, è una pratica che promuove la creazione di un sistema automatico che sappia recepire le modifica pubblicate sul sistema di versionamento con lo scopo di produrre un artefatto **potenzialmente** distribuibile in produzione.

Soprattutto nelle grandi organizzazioni, la parola "potenziale" diviene fondamentale perché, seppur l'artefatto sia pronto per essere distribuito, ci possono essere svariati motivi per decidere di non farlo (es: motivi legati al business). Gli obiettivi della continuous delivery variano da caso a caso, e dipendono da tutti quelle attività che un'organizzazione definisce per considerare un pacchetto pronto per gli ambienti di produzione. Per quanto le attività possano essere automatizzate, il passo che determina il rilascio effettivo dell'artefatto rimane in carico alle decisione umana, e quindi manuale.

Da considerare inoltre che, nel ciclo di vita di software di grandi dimensioni (es: grandi monoliti), diviene impossibile applicare tutta la catena di testing e QA per ogni modifica al codice sorgente. Si sceglie quindi di effettuare test su campioni di commit (e quindi raggruppandole), piuttosto che su ognuna. Ad esempio un'azienda può trattare ogni pacchetto come potenzialmente distribuibile, ma pianificare l'effettiva installazione su base settimanale e intervenire manualmente in caso di hotfix applicando gli artefatti frutto della CDE.

I benefici dell'adozione di questa pratica spaziano dalla riduzione del rischio durante la fase rilascio all' aumento dell'efficienza, riduzione del TTM[5] e maggior soddisfazione del cliente finale [2].

1.5.6 Continuous Deployment

Conosciuta come CD, è una pratica DevOps che prevede la messa in produzione automatica e continua di tutte le modifiche integrate dagli sviluppatori.

[5]TTM: Time To Market

Quello che differenzia la CD dalla CDE sono gli obiettivi. La CD ha come obiettivo quello di rilasciare in produzione non appena uno sviluppatore integra una modifica, automaticamente e senza intervento umano [25]. E' fondamentale evidenziare che non esiste CD senza l'applicazione della CDE, seppur non valga invece il contrario. Agire automaticamente sugli ambiente di produzione, significa interagire con l'utente finale. E' necessario quindi adottare delle metodologie che minimizzano il rischio di fallimento:

- adozione di ambienti per il controllo di qualità
- strategie di rilascio
- virtualizzazione basata sui container
- infrastructure as code
- acceptance test (test di accettazione, collaudo)

Ambienti per il controllo di qualità

Consistono in ambienti, destinati a tipologie di utenze differenti, in cui è possibile avere una visione integrata dell'applicazione prima che raggiunga il contesto di produzione.

Gli sviluppatori hanno bisogno di un posto in cui sviluppare e testare il proprio lavoro con quello del resto del team. Gli incaricati del controllo qualità hanno bisogno di un ambiente per testare sia le nuove funzionalità sia per controllare eventuali regressioni delle funzionalità esistenti. La direzione e altro personale devono accedere per revisioni, dimostrazioni, informazioni di marketing e documentazione.

Un ambiente di questo tipo include sia l'hardware fisico che i componenti software per consentire all'applicazione di funzionare correttamente. Gli ambienti in genere sono una *sandbox*, ovvero un ambiente isolato, consistente e indipendente, in grado di garantire l'interazione dei componenti con un sistema che assomigli il più possibile a quello predisposto per produzione.

Diviene dunque fondamentale nel ciclo di vita dello sviluppo del software, l'aggiornamento e la manutenzione di questi ambienti applicativi.

Gli ambienti :

- **Locale**: Gli sviluppatori creano e testano il codice sulle proprie macchine. Nel caso si necessiti di testare l'integrazione verso servizi terzi gli sviluppatori possono:
  - utilizzare servizi *finti* (mock) per simulare altri pezzi di software come un database o un servizio web
  - utilizzare servizi reali ma disponibili su macchine terze (es: server)
  - utilizzare servizi reali installati sul proprio computer
  - utilizzare servizi reali eseguiti tramite **container** in un contesto applicativo che simula l'ambiente di produzione (es: docker-compose)
- **Sviluppo**: è l'ambiente nel quale gli sviluppatori possono controllare l'integrazione tra tutte le parti del sistema, in genere aggiornato all'ultima commit sul ramo di sviluppo.
- **Staging/ QA / Pre-Produzione**: Sono ambienti dove avvengono la maggior parte dei test per il controllo di qualità, sicurezza e UI/UX[6]. L'infrastruttura rispecchia il più fedelmente possibile l'ambiente di produzione.

[6]User Interface e User Experience: relativamente interfaccia utente ed esperienza utente

- **Produzione**: ambiente finale che ospita la versione del software destinata all'utente finale.

In base al flusso di lavoro scelto per il proprio team, si possono adottare diverse politiche con la quale mantenere gli ambienti aggiornati. In genere si legano a flussi di CI/CD scatenati su rami di sviluppo predefiniti. Ad esempio un modello molto adottato è quello di avere tre rami: *development, test, master*. Effettuare una qualsiasi modifica su tali rami innescherà il processo di CD/CD che aggiornerà, in caso di successo, rispettivamente l'ambiente di sviluppo, test e produzione.

Strategie di rilascio

- **Blue-Green Deployment**: consiste nell'effettuare il rilascio degli aggiornamenti in un ambiente di produzione parallelo e identico all'originale. Quando i test di accettazione sull'ambiente parallelo terminano con successo, si istruisce un bilanciatore di carico per migrare il traffico dal vecchio ambiente a quello nuovo. Questo strategia ci permette di reagire velocemente a problemi riscontrati nell'aggiornamento: basterà infatti redirigere nuovamente il traffico verso l'ambiente che contiene ancora la versione vecchia.
- **Canary Deployment**: il rilascio avviene gradualmente, a gruppi di utenti ristretto. Due tipologie di approccio: deviando una percentuale di traffico prefissata, oppure scegliendo il bacino di utenza basata su delle caratteristiche note (es: sesso, età, paese, eccetera).
- **Rolling Deployment**: strategia di rilascio che consiste nel rimpiazzare gradualmente le istanze di produzione con delle nuove istanze contenenti la versione aggiornata. Questa tecnica permette di evitare interruzioni nell'erogazione dei servizi.

Virtualizzazione basata su container

I *container* sono una soluzione al problema di come far funzionare il software in modo affidabile quando viene spostato da un ambiente di elaborazione ad un altro. Questo potrebbe essere dal portatile di uno sviluppatore a un ambiente di test, da un ambiente di staging alla produzione, da una macchina fisica in un data center a una macchina virtuale in un cloud privato o pubblico. I problemi sorgono quando l'ambiente di esecuzione non è identico: ad esempio sviluppando su Python 2.7, ma trovandosi la versione 3 in produzione. Oppure affidarti al comportamento di una certa versione di una libreria SSL e trovarsi in produzione tutt'altra implementazione. Si hanno problemi frequenti anche quando si adottano sistemi operativi diversi tra macchina sviluppo e ambiente produzione (es: locale su una distribuzione Debian mentre la produzione è su Red Hat). E non è solo un software diverso che può causare problemi: ad esempio si possono avere incongruenze con la topologia di rete, oppure i criteri di sicurezza o le tecnologie di archiviazione.

I container risolvono queste problematiche perché costituiti da un intero ambiente di esecuzione: l'applicazione, più tutte le sue dipendenze, librerie e altri file binari e file di configurazione necessari per eseguirlo, raggruppati in un unico pacchetto. Containerizzando la piattaforma dell'applicazione e le sue dipendenze si promuove la **consistenza**: le differenze nelle distribuzioni del sistema operativo e nell'infrastruttura sottostante vengono eliminate.

Con la tecnologia di virtualizzazione classica, l'unità primitiva di lavoro è la macchina virtuale, che include un intero sistema operativo e l'applicazione. Un server fisico che esegue tre macchine virtuali avrebbe un hypervisor e tre sistemi operativi separati in esecuzione su di esso.

Al contrario, un server che esegue tre applicazioni containerizzate con Docker esegue un unico sistema operativo e ogni container viene eseguito come un processo isolato nello user space. Questo processo condivide il kernel del sistema operativo con gli altri container. Le parti condivise del sistema operativo sono di sola lettura, mentre ogni container ha il proprio mount (cioè un modo per accedere al container) per la scrittura. Ciò significa che i container sono molto più leggeri e utilizzano molte meno risorse rispetto alle macchine virtuali.

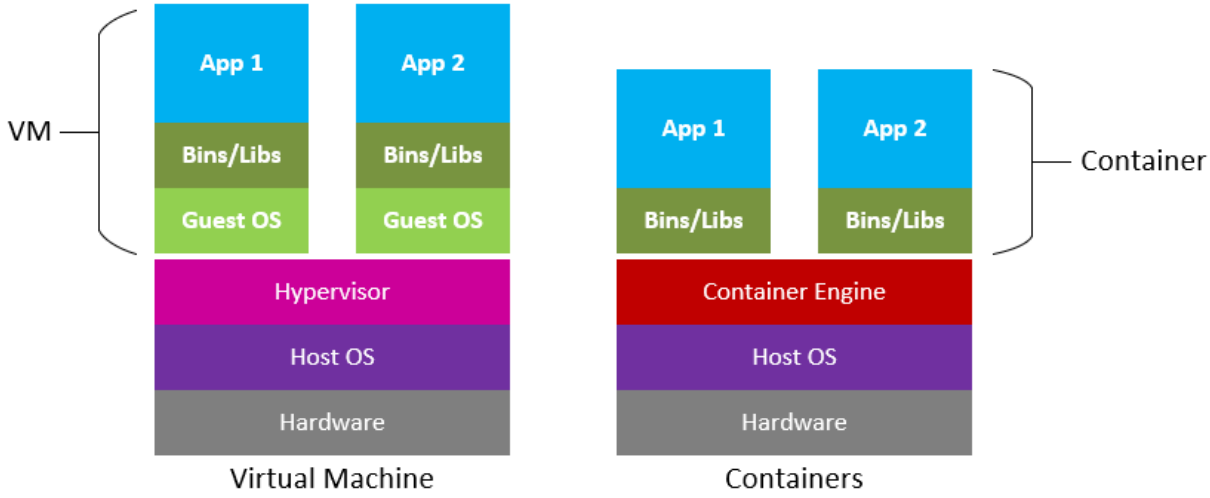

Figura 1.2: Confronto tra VM e Container

Un container può avere una dimensione di solo decine di megabyte, mentre una macchina virtuale con il proprio intero sistema operativo può avere dimensioni di diversi gigabyte. Per questo motivo, un singolo server può ospitare molti più container rispetto alle macchine virtuali.

Un altro vantaggio importante è che le macchine virtuali possono impiegare diversi minuti per avviare i loro sistemi operativi e iniziare a eseguire le applicazioni che ospitano, mentre le applicazioni containerizzate possono essere avviate quasi istantaneamente. Ciò significa che i container possono essere istanziati in modo "just in time" quando sono necessari e possono scomparire quando non sono più necessari, liberando risorse sui loro host.

Un terzo vantaggio è che la containerizzazione consente una maggiore modularità. Anziché eseguire un'intera applicazione complessa all'interno di un singolo container, l'applicazione può essere suddivisa in moduli (come il database, il front-end dell'applicazione e così via). Questo è il cosiddetto approccio dei microservizi. Le applicazioni create in questo modo sono più facili da gestire perché ogni modulo è relativamente semplice e le modifiche possono essere apportate ai moduli senza dover ricostruire l'intera applicazione. Poiché i container sono così leggeri (e reattivi nell'avviarsi), è possibile creare repliche di istanze di singoli moduli (o microservizi) solo quando effettivamente necessari.

Le proprietà principali dei container portano ad una serie di vantaggi anche dal punto di vista dello sviluppo locale:

- Accesso a nuove tecnologie: l'avvio di un'applicazione si riduce al tempo necessario per scaricare la relativa immagine del container. Questo permette la gestione di ambienti di sviluppo o testing con stack tecnologici avanzati, senza preoccuparsi di conflitti o di come si installa quel particolare software nel mio sistema operativo di riferimento.
- Ambienti di sviluppo ben isolati: lavorare con progetti che sfruttano tecnologie non compatibili non sarà più un problema. Non è più necessario programmare script o librerie per la gestione di runtime incompatibili (es: python 2 e 3). Un container, una volta distrutto, non lascia tracce nel sistema operativo che l'ha eseguito.
- Riduzione del tempo di inserimento per colleghi che si approcciano per la prima volta ad alcune tecnologie.
- I container e l'eventuale orchestrazione è descritta da file testuale. Di conseguenza sarà possibile far evolvere l'ambiente di sviluppo versionandola insieme alla base di codice dell'applicazione.

Infrastructure as code

La progettazione dell'infrastruttura è la fase del ciclo di vita del software che definisce e configura tutte le componenti di sistema necessarie per il corretto funzionamento dell'applicazione. In genere queste componenti sono di tipo hardware, software e rete. La tipologia, insieme alla modalità di fruizione (Cloud Provider tramite modelli IaaS, PaaS, SaaS oppure da un fornitore privato), determina le modalità di utilizzo e configurazione.

A livello operativo la gestione della parte infrastrutturale spesso si riduce ad un'orchestrazione, spesso manuale, di scripts che permettono di: (a) istanziare, configurare e collegare tra loro le macchine richieste (sia fisiche che virtuali) per l'esecuzione del software; (b) istanziare ed eseguire i servizi ausiliari necessari per il funzionamento del nostro software (es: database, middleware di comunicazione, eccetera); (c) installare e configurare il software per utilizzare i servizi precedentemente creati [3, 4].

In questo frangente DevOps promuove un insieme di pratiche che utilizzano *codice* (piuttosto che comandi manuali) per la configurazione di macchine e reti (virtuali), installazione di pacchetti e configurazione dell'ambiente per le applicazioni da rilasciare [6]. Queste pratiche prendono il nome di **Infrastracture as Code, IAC.**

L'infrastruttura gestita da questo codice include sia componenti fisiche come apparecchiature (*bare metal*), sia macchine virtuali, containers e reti definite dal software.

Il codice generato per la pratica di IaC deve essere sviluppato e gestito utilizzando gli stessi processi di qualsiasi altro software; per esempio, dovrebbe essere progettato, testato e archiviato in un repository di controllo di versione (come git).

Sebbene gli operatori del settore IT abbiano sempre adottato metodologie per codificare e automatizzare la gestione dell'infrastruttura tramite scripts ad hoc, le pratiche di IaC si sono formalizzate ed hanno preso piede con la diffusione su larga scala del cloud computing, e in particolare della tecnologia IaaS (Infrastructure as a Service). In un ambiente basato su IaaS, tutte le risorse di calcolo, archiviazione e rete sono virtualizzate e

devono essere allocate e configurate utilizzando le interfacce di programmazione fornite dal cloud provider (API). Seppur la quasi totalità dei cloud provider fornisca uno strato di UI, basata su queste API, la gestione diventa scomoda non appena si raggiunge un numero non banale di componenti da istanziare. Ad esempio la creazione di una nuova macchina virtuale (VM) attraverso la console di gestione di Amazon Web Services (AWS) richiede un passaggio attraverso almeno cinque schede di input e la compilazione di circa 25 campi. Utilizzando le pratiche di IaC, si risolve questo problema utilizzando strumenti che generalmente comprendono un linguaggio di scripting per definire dichiarativamente la configurazione di sistema desiderata e un motore di orchestrazione che esegue gli script e richiama l'API IaaS per la creazione delle componenti necessarie.

Adottare IaC significa introdurre vantaggi come come:

- file di testo programmabili e parametrici: l'infrastruttura è versionata insieme al codice sorgente
- la conoscenza della totalità dell'infrastruttura viene esplicitata su un file piuttosto che distribuita nelle competenze di più persone
- utilizzo di uno stile dichiarativo piuttosto che imperativo: i file descrivono lo stato desiderato e non come arrivare in tale stato (a differenza dei vecchi script di codice)
- creazione rapida e automatica di ambienti simil produzione dove eseguire test e prove di carico (oppure per applicazione di blue-green deployment)
- riduzione degli errori dovuti alla gestione manuale dell'intero stack applicativo

1.5.7 Continuous Monitoring

Introdurre un approccio di rilascio rapido combinato a processi di cambiamento continuo e automatici, non può che portare ad affrontare nuove problematiche.

L'intero processo, dalle attività di commit al rilascio in produzione con relativo analisi dello stato dell'infrastruttura, deve essere attentamente esaminato e controllato. Trascurare anche solo un elemento della pipeline può invalidare l'intero processo con conseguente interruzione del servizio, insoddisfazione del cliente e perdita di qualità.

Sul principio DevOps del "misura tutto", nascono soluzione di CM (Continuous Monitoring and Observability) che permettono di svolgere le seguenti attività:

- Analisi delle tendenze a lungo termine: quante build sto eseguendo ogni giorno? Quante ne ho effettuate un mese fa? Devo potenziare o adattare la mia infrastruttura?
- Analisi delle anomalie: come procedono il carico della CPU, l'allocazione della RAM, le statistiche sul traffico di rete, il consumo di memoria e la disponibilità di spazio libero su disco. Tutti i nodi sono attivi?
- Confronto storico: il mio ciclo di rilascio è più lento della settimana scorsa? O è più veloce?
- Scansioni delle vulnerabilità: il mio codice introduce errori software critici e vulnerabilità di sicurezza come perdite di memoria, variabili non inizializzate, limiti di array? Quanto velocemente vengono rilevati? Quanto velocemente vengono risolti?
- Avvisi: qualcosa si è rotto o potrebbe rompersi presto? La mia pipeline di test è stata superata? Devo eseguire un ripristino rispetto al mio ultimo rilascio?

- Condurre un'analisi di correlazione: la mia latenza è appena aumentata, cos'altro è successo nello stesso periodo? Questi eventi sono collegati tra loro?

Il monitoraggio e l'osservabilità continui sono da considerare come la spina dorsale di un sistema di rilascio automatizzato, poiché aiutano a garantire l'integrità, le prestazioni e l'affidabilità delle applicazioni e dell'infrastruttura in ogni fase del ciclo di vita delle operazioni DevOps e IT.

Monitoraggio e osservabilità

Precedentemente si è parlato sia di monitoraggio sia di osservabilità. I due termini non sono da considerare sinonimi ma complementari, in quanto mirano a risolvere obiettivi differenti.

Il **monitoraggio** consiste nel collezionare, processare, aggregare e visualizzare dati quantitativi riguardo parametri significativi del sistema. Lo scopo è quello di ottenere visibilità sullo stato del sistema con l'obiettivo di reagire il prima possibile a situazioni critiche. Questi dati prendono il nome di metriche e hanno diversi formati:

- risultati di interrogazioni su metriche (cpu, ram, latenza, eccetera)
- numero e tipologia di errori, stacktrace
- tempo di esecuzione, eventi, log, tempo di funzionamento (detto anche uptime)

L'importanza del monitoraggio consiste nella capacità di tradurre questi dati e metriche in conoscenza sull'esperienza utente degli utilizzatori del sistema. Avere queste informazioni è fondamentale per re-ingegnerizzare i processi con lo scopo di fornire valore aggiunto.

L'**osservabilità** è una proprietà che consente di diagnosticare lo stato interno del sistema a partire dalla conoscenza degli output attesi. Per tale scopo il sistema dovrebbe essere progettato, costruito, testato, implementato, gestito, monitorato, mantenuto ed evoluto tenendo presenti i seguenti presupposti:

- Nessun sistema complesso è mai completamente sano.
- I sistemi distribuiti sono imprevedibili.
- È impossibile prevedere tutti i possibili guasti che il sistema potrebbe subire.
- I guasti devono essere considerati come la normalità e bisogna aspettarseli in ogni fase del ciclo di vita software.
- La facilità di debug è il principio alla base della manutenzione e dell'evoluzione di sistemi robusti.

Automazione

Poiché l'automazione è uno degli ingredienti chiave di un'efficiente ciclo di rilascio, ha perfettamente senso includere in tale automazione anche il monitoraggio e l'osservabilità nello stesso modo in cui sono stati automatizzati integrazione, test e distribuzione. Per ottenere ciò, in genere è necessario arricchire lo stack tecnologico con tecnologie in grado di effettuare:

- Monitoraggio automatico dei componenti dell'applicazione: processo attraverso il quale il codice dell'applicazione viene esteso per acquisire dati di telemetria per le operazioni di interesse. La raccolta automatica viene comunemente implementata

aggiungendo middleware che si innestano attorno alcuni pezzi di codice significativi con la logica di estrazione della telemetria. Questo ti dà la possibilità di intervenire su qualunque tipo di servizio senza bisogno di modificare manualmente tutte le parti interessate.
- Avvisi automatici: Un meccanismo di segnalazione efficiente basato sui dati di telemetria raccolti. Utilizzano regole per generare avvisi, rilevare errori o condizioni anomale e combinano avvisi con webhook[7] in grado di scatenare azioni utili alla risoluzione del problema.

1.5.8 Pipeline di CI/CD

Una pipeline di CI/CD automatizzata consiste in un unico sistema che combina integrazione, consegna e distribuzione continua. Questo sistema è completamente automatizzato in modo che non richieda intervento umano quando viene attivato. Gli attivatori non sono altro che eventi emessi da attività svolte sulla piattaforma di versionamento e al quale è possibile reagire per innescare una pipeline: ad esempio un commit, una merge request o la pubblicazione di un nuovo tag.

Con *pipeline* si intende una serie di passi che il codice sorgente deve eseguire per completare un'iterazione completa del ciclo di rilascio. Ad ogni passo subisce una verifica o una raffinazione fino a diventare il pacchetto finale di rilascio ed essere in grado di essere applicato, come nuova versione, nell'ambiente di destinazione. L'utilizzo di una pipeline automatizzata può ridurre i costi ed eliminare gli errori umani. Inoltre può fornire anche feedback importanti sul codice, punti di controllo per il superamento di soglie di qualità, distribuzione rapida agli utenti finali e opportunità per gli sviluppatori di rimuovere costi legati ad attività superflue e di concentrarsi sulla scrittura di software di qualità.

Una pratica introdotta nei sistemi di integrazione moderni è quella di descrivere le pipeline in termini di specifica testuale (*pipeline as code*). In questo modo *(i)* lo sviluppatore non ha bisogno di lasciare il proprio ambiente di sviluppo per programmare manualmente un sistema terzo; *(ii)* le pipeline possono essere descritte insieme al codice sorgente e di conseguenza evolvere insieme all'applicazione stessa.

Non esiste una formulazione standard che possa incapsulare tutte le sfaccettature e le necessità di un'applicazione e di un team. Le variabili che determinano i passi necessari dipendono da fattori come l'applicazione stessa, l'infrastruttura in cui risiede, le necessità del team e dell'azienda in cui opera. Di seguito invece che enunciare i passi, riporteremo un elenco generico di azioni, più o meno opzionali, che entrano in gioco durante l'esecuzione della pipeline di CI/CD:

1. Commit sulla piattaforma di versionamento
2. Compilazione
3. Test (smoke test, unit test, integration test, end to end, eccetera)
4. Analisi statica del codice (scansione bug / cve[8] / stile del codice)
5. Creazione pacchetto di produzione
6. Creazione / Validazione Infrastruttura produzione
7. Applicazione delle configurazioni

[7] Sono delle callback HTTP definibili dall'utente ed eseguite al verificarsi di specifici eventi
[8] Common Vulnerabilities and Exposures (CVE), `https://cve.mitre.org/`

8. Installazione del pacchetto (ambiente test / qa / produzione)

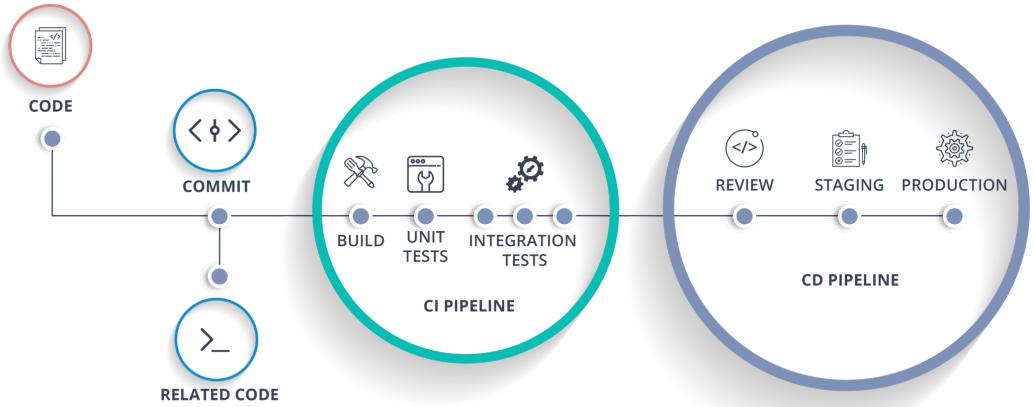

Figura 1.3: Un esempio di Pipeline

1.5.9 Oltre DevOps: altre realtà emergenti

DevSecOps

Estende DevOps affiancando la sicurezza ad ogni fase del ciclo di vita DevOps. Si supera il metodo tradizionale in cui i test di penetrazione e la valutazione delle vulnerabilità vengono eseguiti nelle fasi finali del ciclo di vita del software. La mentalità alla base di questo movimento è "tutti sono responsabili della sicurezza". Questo implica l'inserimento di pratiche di sicurezza nella pipeline DevOps di un'organizzazione.

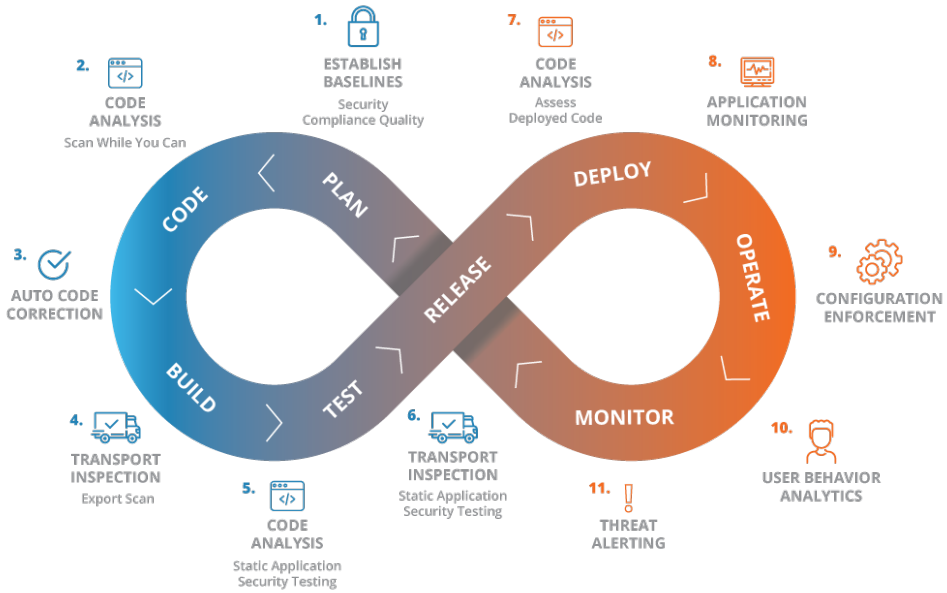

Figura 1.4: DevSecOps, Fonte - Onapsis

GitOps

GitOps nasce come un'evoluzione di Infrastructure as Code (IaC) e una pratica DevOps che sfrutta Git come unica fonte di verità e meccanismo di controllo per la creazione,

l'aggiornamento e l'eliminazione dell'architettura di sistema. Più semplicemente, si basa sul meccanismo che utilizza le pull-request di Git per testare e distribuire automaticamente le modifiche all'infrastruttura del sistema. GitOps garantisce che l'infrastruttura cloud di un sistema sia immediatamente riproducibile in base allo stato di un repository Git. Le pull-request modificano lo stato del repository Git. Una volta approvate e unite nel ramo principale, piattaforme specializzate riconfigureranno e sincronizzeranno automaticamente l'infrastruttura con il nuovo stato del repository.

MLOps

MLOps (un composto di Machine Learning e "Information technology OPerationS") da un lato adotta ed estende i princìpi DevOps, promuovendo la collaborazione e la comunicazione tra data scientist e professionisti del settore IT, dall'altro ha come obiettivo l'automazione del ciclo di vita della produzione di algoritmi e modelli di machine learning. Attraverso pratiche e strumenti, MLOps mira a stabilire una cultura e un ambiente in cui le tecnologie ML possono generare vantaggi per il business attraverso costruzione, testing e rilascio della tecnologia ML in modo rapido, frequente e affidabile.

NoOps

NoOps significa nessuna operazione. La filosofia alla base consiste nel rimuovere tutte le parti di gestione della piattaforma e ridurre l'attrito tra sviluppatori e infrastruttura. Lo scopo di NoOps è definire una serie di processi in cui non sia più necessario combinare la parte dello sviluppo con quella operativa. L'obiettivo di NoOps è quello di adottare processi snelli e completamente automatici che riducono la necessità di avere un team dedicato alla pratiche operative e gestione del software *in-house*. Fondamentalmente, un approccio NoOps è il seguente: lo sviluppatore salva il codice sulla piattaforma di versionamento e tutto il ciclo di rilascio avviene in modo automatico. NoOps è possibile grazie alla nascita di piattaforme abilitanti (es: PaaS[9] CI/CD, Computazione Serverless).

[9]Platform as a Service (PaaS), con cui ci si svincola dalla gestione dell'infrastruttura sottostante (hardware e sistemi operativi) in modo da concentrarsi sulla distribuzione e sulla gestione delle applicazioni

Capitolo 2

Caso di studio: Sistema Editoriale di Maggioli

2.1 Introduzione

L'editoria digitale necessita di canali attraverso cui fornire i contenuti prodotti dagli autori. In Maggioli S.p.A.[1] un enorme passo avanti in questa direzione è stato fatto aprendosi ad un modello di vendita digitale supportato da canali di web marketing che sfruttano una pletora di portali aperti o in abbonamento per ottenere la massima visibilità e divulgazione.

Il progetto interno, chiamato *sistema redazionale* (nome in codice *sisred*), nasce circa 15 anni fa per semplificare le modalità di fruizione di tutti i contenuti editoriali (libri, riviste, contenuti web, eccetera) da parte di un insieme di servizi come motori di ricerca avanzati, applicazioni per professionisti, portali web specializzati ed applicazioni mobile. Il sistema redazionale ha due obiettivi principali: *(i)* fornire supporto ai creatori di contenuto (redattori); *(ii)* essere il mezzo che veicola questi contenuti verso l'utente finale.

2.2 Panoramica generale

Il software originale consiste in una classica applicazione client-server orientata verso professionisti del diritto. Gli editori sono esperti di dominio pagati da Maggioli che agiscono come fonti di informazioni e viene fornita un'installazione locale di un client grafico che consente loro di aggiungere contenuti editoriali a un database centrale. Tipici esempi di questi contenuti sono leggi nazionali o regionali, documenti legalmente vincolanti di istituzioni pubbliche (ad esempio, ministeri) e frasi.

Le modifiche vengono quindi propagate verso destinazioni finali, chiamati prodotti, che consistono solitamente in siti Web o motori di ricerca basati su WordPress, il cui accesso è determinato dalla sottoscrizione di un contratto di abbonamento. La propagazione può essere innescata manualmente dagli editori, ma di solito è lasciato a un processo di esportazione notturno.

[1] `https://www.maggioli.it/`

2.2.1 Architettura originale

L'architettura di riferimento è illustrata nella fig. 2.1. L'implementazione del client grafico è in Delphi, mentre la logica di esportazione è scritta nei file batch di Windows. Il database centrale che contiene i contenuti di tutti i prodotti è un database relazionale MS-SQLServer, che contiene oltre ai dati dello strato di persistenza, anche logica di business materializzata sotto forma di *stored procedure*.

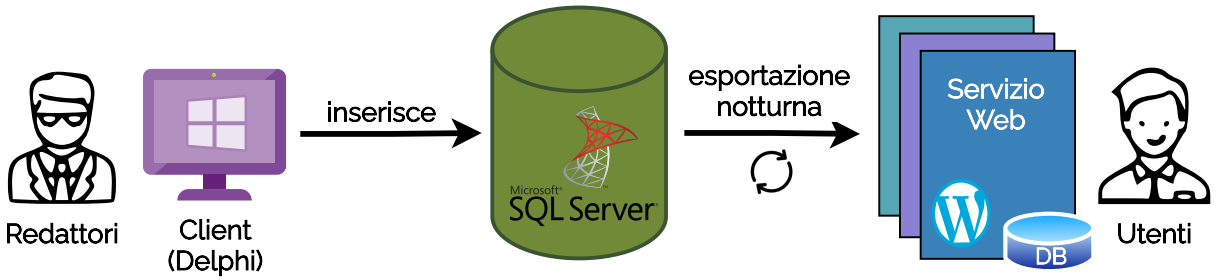

Figura 2.1: Architettura Sisred

2.2.2 Contenuti del sistema

A causa della natura di business del settore di Maggioli Editore, i contenuti digitali non hanno una natura generica, ma anzi altamente specializzata verso tipologie di professionisti come Avvocati, Ingegneri, Architetti, Commercialisti, eccetera. Di conseguenza il dominio secondo cui vengono classificati i contenuti editoriale è modellato come segue:

- Normativa: si tratta di leggi.
- Prassi: procedure riferite ad attività regolate solo da norme generali e non codificata in una legge. Ad esempio circolari, determine, delibere e comunicati.
- Giurisprudenza: sono sentenze dei tribunali. Sono precedute sempre da una "massima" che rappresenta un commento introduttivo di un redattore.
- Dottrina: sono approfondimenti redatti dagli autori.
- Profor: Tutto ciò che non rientra nelle categorie precedenti (ad esempio modulistica)

2.2.3 Inserimento e pubblicazione dei contenuti

Tramite l'interfaccia grafica del sistema redazionale, il redattore può inserire nuovi contenuti nel sistema. L'inserimento avviene tramite schermate che sono specifiche per le categorie di contenuto da inserire (perché sono diversi i metadati che lo descrivono). I contenuti sono inseriti con gli estremi di emanazione o pubblicazione (ente emittente, data, numero di provvedimento, oggetto, sede, eccetera) oltre al testo vero e proprio del contenuto. In base alla categoria del documento è predisposta una maschera di inserimento specifica. In Figura 2.2 un esempio di form di inserimento.

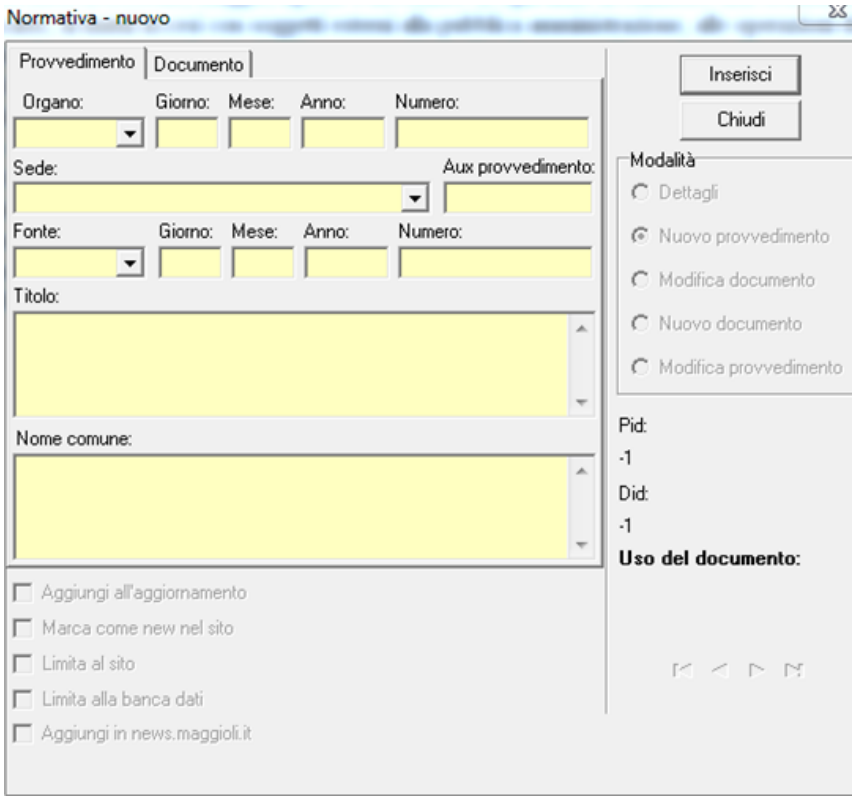

Figura 2.2: Inserimento di inserimento di un documento di normativa

Per quanto riguarda le fonti che danno origine ad un contenuto:

- Normativa: Gazzetta Ufficiale per le norme nazionali oppure Bollettini regionali per le norme locali
- Prassi: Siti delle autorità e degli enti emittenti (ANAC, AGCM, Ministeri)
- Giurisprudenza: i testi integrale delle sentenze del TAR e del Consiglio di Stato vengono presi dal sito della giustizia amministrativa[2]. Le massime sono invece un commento dei redattori frutto della loro interpretazione della sentenza.
- Dottrina: tutti i contenuti di questa categoria vengono scritti dai redattori.

Una volta inserito nella base documentale, il redattore può organizzare in quale prodotto (portale wordpress, motore di ricerca, mobile, CD da allegare ad un libro) deve essere pubblicato. La pubblicazione può essere immediata o asincrona (via processi automatici di esportazione notturna).

2.3 Organizzazione dei processi

Il team incaricato di mantenere il sistema redazionale consiste di due sviluppatori e un responsabile di progetto. Di seguito si formalizza l'organizzazione dei processi e le eventuali problematiche correlate, frutto di un'intervista a tale team.

[2] `https://www.giustizia-amministrativa.it/`

2.3.1 Gestione delle attività

La gestione delle attività è compito del responsabile di progetto. Non esiste una piattaforma in cui gli sviluppatori possono controllare e contribuire alla programmazione delle richieste o all'evoluzione delle priorità di sviluppo. Tutte le attività sono state organizzate manualmente dal responsabile del team e provenivano da due fonti principali: richieste dirette tramite posta elettronica e ticket segnalati tramite un servizio di terze parti[3]. L'assenza di una piattaforma non permetteva di avere diretta corrispondenza tra problemi/e-mail e relative modifiche fatte sulla base di codice. Nonostante il processo sia largamente migliorabile, nessuno dei membri del team ha sollevato specifiche problematiche; i motivi si identificato in una buona comunicazione a livello di team, favorita dalla piccola dimensione della squadra e dal fatto di lavorare fisicamente in postazioni di lavoro a stretto contatto. Una piccola lamentela riguarda l'uso obbligatorio di Skype come politica aziendale per lo strumento di comunicazione remoto. Vedremo successivamente la direttiva rimossa in base alla riorganizzazione aziendale promossa in risposta alla pandemia COVID-19.

2.3.2 Controllo di versione

L'intera applicazione risulta parzialmente versionata in base alle diverse componenti che la compongono:

- Client Delphi: I sorgenti della parte client sono interamente versionati su svn. Non è implementata alcuna politica di branching; ogni sviluppatore lavora parallelamente e direttamente sul ramo principale *trunk*. I commenti dei commit sono liberi mentre per il versionamento è adottata una numerazione incrementale.
- Database e stored procedure: entrambi non versionati. Gli aggiornamenti al database sono scritti in locale e poi lanciati su ambiente test e produzione tramite il client di gestione del database (SqlServer Management Studio). Le stored procedure sono progettate ed eseguite direttamente sul SQLServer di test e successivamente riportate in produzione quando ritenute funzionanti.
- File batch: i file di scripting per l'esportazione verso i prodotti non sono versionati. La modifica avviene direttamente sui server degli ambienti di test e produzione, previa richiesta verso il team operativo.

Problematiche emerse: sistema di controllo di versione usato solo come backup dei file sorgenti. Difficoltà di integrazione delle modifiche dopo giorni di lavoro in ambiente locale.

2.3.3 Ambiente di sviluppo

Gli strumenti utilizzati per sviluppatore in locale:

- Delphi IDE: per la scrittura della parte Delphi e testing della UI/UX
- SQL Server Management Studio: per la gestione della proprio istanza di SQL Server 2000. Qui sopra avviene la modellazione del db in locale e il testing delle stored procedure

[3] `https://osticket.com/`

Problematiche emerse: installazione di SQL Server può causare problematiche a seconda della versione del sistema operativo dello sviluppatore. Impossibilità di sviluppare su sistemi operativi diversi da Windows.

2.3.4 Ambienti staging e produzione

Nel caso di necessità di un nuovo ambiente che rispecchi produzione, occorre inviare richiesta via email al team dei sistemisti. Dopo aver evaso la richiesta sarà disponibile un server con Sistema Operativo Windows e Microsoft SQLServer installato. Successivamente i sistemisti contatteranno il team di sviluppo per ricevere e installare l'ultima versione dello schema DB e degli script di esportazione.

Problematiche emerse: tanti tempi morti e colli di bottiglia nella comunicazione tra team di sviluppo e team sistemisti.

2.3.5 Compilazione e distribuzione aggiornamenti

Il flusso per la distribuzione di aggiornamenti è il seguente:

1. Commit delle modifiche su SVN.
2. Creazione exe con versione assegnata manualmente da Delphi IDE.
3. Aggiornamento manuale, in ambiente di test, del database (Schema, Stored Procedure).
4. Aggiornamento manuale, in ambiente di test, dei file di scripting per l'esportazione dei contenuti verso i prodotti.
5. Il team effettua test manuali sull'ambiente di staging (Controllo funzionalità e UI).
6. Il team dei sistemisti riportano gli aggiornamenti in produzione.
7. Invio del nuovo client ai redattori (via email).

Problematiche emerse: molto tempo degli sviluppatori viene investito in attività automatizzabili. La distribuzione dei pacchetti necessità dell'intervento dei sistemisti. L'invio del client è fatto attraverso email.

2.3.6 Gestione dei test

I test sono gestiti manualmente dal team di sviluppo. In primo luogo sulla loro macchina di sviluppo, in secondo luogo sull'ambiente di staging. Nel caso di riscontro di problematiche lo sviluppatore è tenuto a segnalarlo a tutto il team. Tuttavia non è presenta una piattaforma che consenta la gestione del ciclo di vita di tale problematica (creazione, assegnazione, priorità, tracciamento dei tempi, scadenze, eccetera).

Problematiche emerse: i test sono effettuati manualmente e dipendono dalla persona che li sta svolgendo. I test in genere provano nuove funzioni e non mirano a cercare eventuali regressioni. Mancanza di una piattaforma dedicata.

2.3.7 Gestione delle problematiche e feedback

Come già visto per l'organizzazione delle attività, in caso di bug o problemi con l'applicazione, la segnalazione è fatta attraverso lo strumento *Osticket*. Il team di sviluppo riceve

una notifica via email e pianifica la risoluzione. La piattaforma di controllo del codice sorgente e quella di gestione dei ticket non hanno interoperabilità applicativa.

Problematiche emerse: non esistono meccanismi in grado di intercettare automaticamente problematiche e regressioni sul software e infrastruttura di produzione.

2.4 Metriche

Di seguito uno studio su metriche significative allo scopo di presentare la situazione attuale rispetto la metodologia di sviluppo e le pratiche adottate. Nel leggere le metriche si tenga conto che quando non diversamente specificato i valori sono intesi come media rispetto all'ultimo anno di sviluppo.

Frequenza dei rilasci in produzione

Indicatore di quante volte viene rilasciato un aggiornamento in produzione.

- Valore medio: 0,071 rilasci al giorno (1 volta ogni due settimane)
- Fonte: intervista al team

Durata ciclo di rilascio

Indicatore del tempo medio impiegato a partire dalla compilazione del pacchetto fino all'effettiva installazione sul client o sui server.

- Valore medio: 1 giorno
- Fonte: intervista al team
- Motivo: Generazione pacchetto, installazione degli aggiornamenti sul server, invio dell'eseguibile della UI ai redattori via email

Tempo di sviluppo di un'idea

Indicatore di quanto tempo impiega un'idea a concretizzarsi nell'ambiente di produzione, passando per tutte le fasi del ciclo di vita del software.

- Valore medio: 14 giorni
- Fonte: intervista al team

Numero di commit al giorno

- Valore: 2
- Fonte: analisi log svn

Tempo medio di recupero (MTTR)

Indica mediamente quanto tempo si impiega per ripristinare produzione dopo una situazione di guasto grave (es: corruzione database, rottura hardware, ecc)

- Valore: 1,5 giorni

- Fonte: intervista al team
- Motivo: Dopo aver appurato il guasto, si contatta il team dei Sistemisti che provvederanno a ripristinare un backup in caso di guasto software, oppure l'intero server in caso di guasto hardware

Tasso di fuga degli errori

Indica mediamente quanti errori superano gli ambienti di test, forzando a intervenire con hotfix subito dopo il rilascio di aggiornamenti.

- Valore: al 50% dei rilasci succede almeno una commit di hotfix
- Fonte: Analisi messaggi di commit svn

Numero di issue/ticket aperti

- Valore: 40 al mese
- Fonte: OSTicket

Tempo di gestione di una issue

- Valore: 4 giorni
- Fonte: portale gestione ticket OSTicket

Tempo di inattività del sistema

- Valore: circa 30 minuti ogni notte
- Fonte: misurazione durata esportazione
- Motivo: i prodotti non hanno i contenuti disponibili durante l'esportazione notturna. Aggiornamento database e client delphi non provocano tempo di inattività del sistema.

Tempo impiegato configurazione ambiente sviluppo

- Valore: 2 ore
- Fonte: dato misurato
- Motivo: download e installazione delphi, installazione SQL Server 2000, configurazione degli IDE

Tempo per la creazione di un ambiente di produzione

- Valore: 2 giorni
- Fonte: dato misurato
- Motivo: si misura il tempo medio rispetto una richiesta inviata al gruppo sistemisti per una macchina server per ospitare SQL Server 2000. La fornitura del server avviene in circa 1,5 giorni più 0,5 giorni per installazione database e script per esportazione notturna.

2.5 Requisiti della trasformazione

In azienda si affronterà il rinnovo funzionale e tecnologico dell'intero progetto. Per la realizzazione della piattaforma sono previsti importanti investimenti su tecnologie particolarmente innovative per la realizzazione di una serie di servizi che forniscono il valore aggiunto collegato al contesto editoriale dell'azienda.

A partire dallo studio dello stato attuale emerso nel capitolo 2, si effettueranno delle interviste alle principali figure che detengono interesse nei confronti della suddetta iniziativa. Queste interviste produrranno dei requisiti che saranno la base su cui si baserà la trasformazione prima dell'intera architettura e, successivamente, dei nuovi processi derivati dall'adozione della filosofia DevOps.

2.5.1 Definizione dei requisiti

Il nuovo sistema redazionale è un progetto interno all'azienda. I requisiti della piattaforma vanno identificati sia in termini di esigenze degli utilizzatori interni (i redattori), sia in termini di opportunità che l'azienda vuole cogliere con l'investimento nella riscrittura del progetto. Di seguito le esigenze raccolte durante le interviste per il rinnovamento del Sistema Redazionale.

Come redattore voglio:

- *R1* mantenere tutte le funzionalità presenti nel vecchio sistema
- *R2* accedere al sistema da qualsiasi dispositivo, a prescindere dal suo sistema operativo
- *R3* una classificazione automatica dei documenti attualmente contenuti (tag, estrazione metadati)
- *R4* la possibilità di gestire altri contenuti multimediali, ad esempio webinar e videocorsi di formazione (streaming video)
- *R5* una funzionalità di ricerca avanzata che agisca su tutta la base dati, a prescindere dal formato del contenuto (testo, immagini, video, eccetera)
- *R6* un sistema di raccomandazioni che possa consigliare contenuti affini a quello in esame
- *R7* integrarmi con flussi automatici di pubblicazione dei contenuti (es: la gazzetta ufficiale della repubblica italiana)
- *R8* rendere i contenuti pubblicati nel sistema redazionale subito visibili senza necessità di aspettare l'esportazione notturna
- *R9* la possibilità di estrarre il testo dalla scansione di un documento
- *R10* uno strumento che mi consenta la creazione di contenuti con riferimenti ad altri documenti della mia base di dati. Ad esempio una sentenza con il riferimento ad un articolo deve essere in grado di ricondurmi all'articolo citato.
- *R11* la possibilità di anonimizzare dati sensibili da documenti che citano nomi e persone

Come azienda voglio:

- *A1* Progettare il nuovo sistema redazionale come un prodotto, in modo da poterlo vendere ad altre aziende del settore. Prima astraendolo quindi dalla specifiche problematiche aziendali, poi introducendo politiche di controllo di qualità e rimuovendo l'interruzione di servizio corrispondente agli aggiornamenti.

A2 Minimizzare TTM[4] e massimizzare la qualità del servizio.
A3 Avere un sistema in grado di reagire rapidamente ai cambiamenti, adattandosi a nuove tecnologie, architetture e condizioni di lavoro.
A4 Abilitare la costruzione di nuovi servizi, sia interni che esterni, che possano sfruttare la base documentale consolidata.
A5 Progettare la nuova infrastruttura in modo da poter sfruttare tutti i vantaggi di un applicazione nativa cloud (self-service, fast startup, elasticità, controllo dei costi).

2.5.2 Vincoli tecnologici

L'esplorazione di nuove tecnologie e strumenti sarà soggetta a vincoli derivanti da politiche aziendali. In particolare:

T1 Sorgenti mantenuti privati e su piattaforma interna *Gitlab CE*.
T2 In caso di utilizzo di soluzioni cloud utilizzare la sottoscrizione Microsoft Azure aziendale.
T3 La ricerca di soluzioni tecnologiche di terze parti, non gestite da un Cloud Provider, deve concentrarsi su versioni open source.

[4]Time to Market

Capitolo 3

Decomposizione in microservizi

3.1 Introduzione

Per raggiungere gli obiettivi descritti in Sezione 2.5.1 occorre applicare un approccio radicale anche nella progettazione dell'architettura dell'applicazione.

I microservizi sono uno stile architetturale nato per la creazione di applicazioni distribuite. Questo stile consiste nel suddividere un'applicazione in piccoli servizi indipendenti, fortemente disaccoppiati e distribuibili individualmente. Ogni microservizio deve essere progettato seguendo il *principio di singola responsabilità*, in cui ogni componente è un'applicazione completa e contiene tutti gli strumenti necessari a portare a termine la funzionalità a lei assegnata. Per tale motivo, una volta stabilita tale funzionalità da raggiungere, viene naturale assegnare la gestione di microservizi a diversi team di sviluppo.

La comunicazione tra le varie sotto-parti determina il comportamento complessivo del sistema. Per questo è necessario definire per ogni servizio un'interfaccia pubblica con la quale comunicare ed esplicitare chiaramente le dipendenze (ad esempio verso altri microservizi o risorse esterne) per far sì che ogni microservizio possa essere rilasciato ed eseguito in modo ragionevolmente indipendente.

Questa architettura consente di scalare o aggiornare le varie parti di sistema senza interrompere le altre funzionalità dell'applicazione. Un framework di microservizi crea un sistema ampiamente scalabile e distribuito, che risolve e supera i limiti di gestione derivati da anni di lavoro su grandissime applicazioni monolitiche, migliorando la capacità di un azienda di fornire un prodotto di maggiore qualità e abilitando pratiche di automazione su stack tecnologici di avanguardia, in linea con i concetti DevOps esposti in capitolo 1. Seppur DevOps e architetture a microservizi possano essere applicati in modo indipendente, il massimo delle sinergia si raggiunge quando vengono combinati insieme. Infatti, da un lato, le pratiche DevOps semplificano la decomposizione e la manutenzione dei sistemi basati su microservizi; d'altra parte la complessa orchestrazione di tali servizi genera un'elaborata sequenza di attività di costruzione, verifica e distribuzione, che beneficiano appieno delle pratiche promosse dall'approccio DevOps.

Il team del nuovo sistema redazionale ha collaborato con due redattori con esperienza decennale, chiamati a partecipare all'analisi come esperti di dominio. La progettazione a

microservizi è stata svolta dapprima analizzando le funzionalità gestite dal software monolita e successivamente integrandole con i requisiti desiderati estratti in sezione 2.5.1. La scomposizione del monolita in microservizi è consistita quindi dai seguenti passi, derivati dall'approccio **Domain-Driven Design, DDD** [11]:

1. Analisi del dominio.
2. Definizione dei bounded contexts: man mano che si modella un dominio si nota come la complessità cresca con il numero di concetti di tale dominio. Un pattern del DDD consiste nel decomporre la visione unificata in tanti ambiti applicativi, chiamati Bounded Context. Concetti dello stesso dominio possono assumere significati diversi in base all'ambito applicativo in cui sono collocati.
3. Definizione di entità, aggregati e servizi per ogni bounded context.
4. Identificazione dei microservizi: a partire dallo studio dei bounded context si organizza un'architettura con diversi servizi, altamente disaccoppiati e che lavorano in modo collaborativo. Questi servizi operano con un'interfaccia pubblica ben definita e devono implementare le funzionalità del business domain seguendo il principio di singola responsabilità.

3.2 Modello di comunicazione

In un'applicazione monolitica in esecuzione in un singolo processo, i componenti si richiamano a vicenda tramite chiamate a metodi o funzioni a livello di linguaggio. Questi componenti possono essere strettamente accoppiati se si creano oggetti con il codice, ad esempio *new ClassName()*, o possono essere richiamati in modo disaccoppiato se si usa la funzionalità di inserimento delle dipendenze facendo riferimento ad astrazioni anziché a istanze di oggetti concrete. In entrambi i casi, gli oggetti vengono eseguiti all'interno del processo stesso.

La sfida maggiore quando si passa da un'applicazione monolitica a un'applicazione basata su microservizi è rappresentata dalla modifica del meccanismo di comunicazione. Non esiste una soluzione che risolva tutti gli scenari e in generali si approccia il problema cercando di mantenere quanto più possibile l'isolamento dei microservizi aziendali. Viene quindi usata una comunicazione asincrona tra i microservizi interni e si sostituisce la comunicazione con granularità fine, tipica della comunicazione tra gli oggetti all'interno di un processo, con una comunicazione con granularità più grossolana che possa esporre i dati che aggregano i risultati di più chiamate interne.

I due protocolli comunemente usati sono il protocollo HTTP con API Rest delle risorse (soprattutto per l'esecuzione di query) e messaggistica asincrona per propagazione eventi e aggiornamento dei modelli tra più microservizi.

- **Protocollo sincrono**. HTTP è un protocollo sincrono. Il client invia una richiesta e attende una risposta dal servizio. Tale processo è indipendente dall'esecuzione del codice client che può essere sincrono (thread bloccato) o asincrono (thread non bloccato e la risposta alla fine raggiunge una funzione di callback). L'aspetto importante è che il protocollo (HTTP/HTTPS) è sincrono e il codice client può continuare l'attività solo quando riceve la risposta del server HTTP.
- **Protocollo asincrono**. Il codice client o il mittente del messaggio in genere non attende una risposta. Invia semplicemente il messaggio analogamente all'invio di

un messaggio a una coda RabbitMQ o a un qualsiasi altro broker di messaggi.

Nel riprogettare il sisred si utilizzerà un approccio ibrido che possa mettere in sinergia i due protocolli individuati. In particolare useremo un protocollo sincrono HTTP quando si vogliono recuperare dati tra un servizio e l'altro, mentre il protocollo asincrono sarà utilizzato per la propagazione di eventi e per impartire comandi che producono processi di esecuzioni di lunga durata. L'eventuale risposta sarà quindi restituita in modo asincrono sul canale di comunicazione.

Per quanto riguarda le specifiche tecnologie, la comunicazione sincrona non prevede l'aggiunta di alcun pezzo infrastrutturale in quanto supportata nativamente da tutti i linguaggi o eventuali librerie. Invece per lo scenario di utilizzo della comunicazione asincrona avremo bisogno di un middleware in grado di offrire garanzia sulla consegna dei messaggi. In questo caso scegliamo di utilizzare un servizio completamente gestito dal nostro Cloud Provider, ovvero Azure Service Bus, aderente allo standard ISO/IEC AMQP 1.0 (Advanced Messaging Queueing Protocol)[1], lo standard di fatto nei sistemi di comunicazione asincrona.

3.3 Metodo di distribuzione

Il processo di rilascio di microservizi introduce complicazioni che in un modello monolitico non erano mai state affrontate. In particolare richiede la capacità di: *(i)* compilare e rilasciare microservizi in modo rapido e indipendente dagli altri dello stesso ecosistema; *(ii)* scalare simultaneamente una moltitudine di servizi, ognuno dei quali sottoposto a carico differente; *(iii)* progettare il reale isolamento: un microservizio deve essere isolato sia in termini logici sia in termini di ambiente di esecuzione. Un fallimento in un microservizio non deve impattare altri servizi fuori dai propri confini.

Come già trattato in Sezione 1.5.6, la metodologia di rilascio a container stende le basi per risolvere parzialmente queste complicazioni. Vedremo successivamente altre pratiche del mondo DevOps che ci consentono di raggiungere la totalità di controllo sul suddetto processo di rilascio, come ad esempio l'automazione, testing e monitoraggio.

Utilizzando Docker, lo standard di fatto, i microservizi possono essere rilasciati seguendo questi passi:

1. racchiudere il microservizio in un immagine di container
2. creazione di un istanza per ogni immagine generata
3. creazione delle reti per il collegamento dei vari container
4. verificare che ogni container sia partito correttamente
5. fornire un bilanciatore di carico che possa smistare il carico tra le diverse istanze di uno stesso microservizio e un service discovery che possa permettere alle singole istanze di registrarsi come fornitrici del medesimo servizio

E successivamente, quando l'applicazione è in esecuzione, rimanere in analisi dello stato del container al fine di: effettuare un eventuale aumento delle risorse o del numero di istanze; oppure accorgersi di un fallimento del container e procedere alla ri-creazione dello stesso.

[1] `https://www.iso.org/standard/64955.html`

È evidente che questo tipo di approccio, oltre che poco pratico, non è sostenibile per un modello a microservizi dove il numero di container da pilotare può spaziare da poche unità a oltre centinaia di istanze. Per tale scopo nascono strumenti che mirano a risolvere queste problematiche effettuando quello che in letteratura è chiamato **Orchestrazione di Container** [24]. Il più famoso in industria prende il nome di Kubernetes (spesso chiamato anche k8s). Ideato da Google e reso pubblico nel 2014. Successivamente mantenuto dal Cloud Native Computing Foundation (CNCF).

3.3.1 Kubernetes

Kubernetes è una piattaforma opensource per la gestione di carichi di lavoro distribuiti su container, servizi e per automatizzare la distribuzione, la scalabilità e orchestrazione di applicazioni containerizzate. Kubernetes non ha limiti rispetto ai diversi paradigmi di progettazione e supporta infatti una varietà di carichi di lavoro come stateful, stateless ed elaborazione dei dati in modalità batch. Se un container è in grado di pacchettizzare ed eseguire un'applicazione, allora anche kubernetes sarà in grado di farlo.

Tutto in Kubernetes è esprimibile *dichiarativamente* tramite oggetti di configurazione che rappresentano lo stato desiderato del sistema. Il compito principale di k8s è quello di garantire che lo stato del mondo reale corrisponda a quest'ultimo. Il paradigma dichiarativo, a differenza di quello imperativo in cui lo stato desiderato si raggiunge tramite l'esecuzione di una serie di istruzioni, permette al programmatore di concentrarsi sul *cosa* debba essere fatto invece che investire tempo nel pensare al *come*. Mentre i comandi imperativi definiscono azioni, le configurazioni dichiarative definiscono lo stato. Sfruttando queste configurazione, il programmatore descrive le risorse che compongono la propria applicazione, effettivamente utilizzando oggetti offerti da kubernetes e che permettono di astrarre le necessità applicative dal mondo in cui è in esecuzione il cluster (per esempio: macchina locale, VM, Cloud Provider, Raspberry, eccetera).

Adottare kubernetes come orchestratore di container significa introdurre nei propri sistemi una serie di funzionalità native che consistono in:

- Gestione risorse e scalabilità automatica: a Kubernetes viene affidata la gestione dell'applicazione e una serie di specifiche riguardo i pesi da associare all'allocazione delle risorse e su quale politica effettuare la scalabilità. Per garantire il completo utilizzo del cluster e risparmiare sulle risorse inutilizzate, Kubernetes effettua un continuo lavoro di bilanciamento tra carichi di lavoro critici e best-effor, scalando in aggiunta o diminuzione delle istanze a seconda dello sforzo a cui è soggetto.
- Bilanciamento del carico (**load-balancer**) e rilevamento dei servizi (**service discovery**): non è necessario preoccuparsi della configurazione e comunicazione attraverso la rete perché lui stesso assegnerà indirizzi IP a ogni container in esecuzione. Inoltre per ogni gruppo logico di istanze che rappresentano un servizio, saranno definiti dei nomi DNS utilizzati per dare un unico punto di accesso al servizio e lasciare a K8s l'onere di effettuare lo smistamento del traffico verso una delle potenziali istanze.
- Orchestrazione dello storage: ad ogni container in esecuzione sul cluster, è possibile assegnare (tramite operazione mount) un sistema di archiviazione preferito. Vi sono varie opzioni di archiviazione: localmente al cluster, servizio di storage cloud oppure uno storage di rete come NFS, iSCSI, eccetera.

- Autoguarigione (self-healing): un componente del cluster è incaricato di monitorare lo stato dei container con lo scopo di rilevare e automaticamente ricreare quelli in stato di fallimento. L'eventuale fallimento critico di un intero nodo provocherà la ridistribuzione del carico sugli altri disponibili.
- Gestione dei **segreti** e della **configurazione**: la gestione dei segreti e delle configurazioni avviene senza bisogno di ricostruire e applicare nuovi container. Questo meccanismo permette di non esporre segreti nella configurazione dello stack.
- Rilascio e ripristino automatici: la distribuzione delle modifiche e aggiornamenti avviene progressivamente, un istanza alla volta, assicurandosi che non via sia interruzione di servizio. Se qualcosa va storto, Kubernetes ripristinerà la situazione precedente.

Agli occhi di un utilizzatore, l'intera architettura di Kubernetes, si presenta come un cluster formato da un numero determinato di nodi organizzati in ruoli master e worker. Lo sviluppatore comunica con le API del master del cluster tramite un client chiamato *kubectl*. L'utilizzo di Kubernetes in una modello architetturale a microservizi consente di trattare la gestione dell'intera applicazione come un unico sistema. Inoltre la capacità di fornire le risorse astraendo dall'effettiva implementazione e hardware, consente alle aziende di eseguire la distribuzione di container rimanendo agnostici dall'effettiva piattaforma di esecuzione (ad esempio non ci si vincola alle singole risorse cloud ma si sfruttano risorse astratte messe a disposizione da k8s). Le distribuzioni di microservizi su larga scala spesso si basano su Kubernetes.

Le funzionalità native di kubernetes viste precedentemente saranno tenute in considerazione nella progettazione dell'architettura in quanto permettono di non introdurre esplicitamente alcuni componenti architetturali come il bilanciatore di carico, service discovery, config server e instradamento del traffico in arrivo da reti esterne (ingress).

3.4 Microservizi identificati

In Sezione 3.1 è stato presentato DDD come modello strategico utilizzato per l'identificazione dei vari microservizi. Dopo l'identificazione del dominio principale e dei sotto-domini, DDD suggerisce di identificare i confini logici tra sotto-contesti: i cosiddetti *bounded-context*. Questi sono fondamentali per la decomposizione del vecchio sistema monolitico in più piccoli servizi indipendenti, in quanto forniscono una segmentazione naturale del dominio originale. Inoltre si raggiunge una visione dell'alto del perimetro di ogni contesto, utile per progettare il coordinamento dei servizi attraverso l'esposizione di interfacce pubbliche per la comunicazione e facilitando quindi il principio di singola responsabilità.

A seconda dei componenti individuati, come già specificato in Sezione 2.5.1, sarà possibile individuare diversi scenari che descrivono il tipo di lavoro che il team dovrà svolgere. Lo scenario è determinato dalla funzionalità che il servizio deve esprimere:

1. Funzionalità specifica del sisred: implementazione completa del microservizio
2. Funzionalità già esistente: integrazione di servizi all'interno dell'ecosistema del sisred
    - servizi distribuiti da altri gruppi aziendali
    - servizi di piattaforma Maggioli

- servizi di piattaforma del cloud provider di riferimento

Di seguito una presentazione dei servizi emersi dalla fase di analisi e che costituiscono una segmentazione ragionevole del modello di dominio originale.

3.4.1 Client Web

Questo servizio sostituisce la vecchia interfaccia dei redattori precedentemente sviluppata in Delphi. La nuova implementazione mira a produrre un'applicazione web di tipo SPA[2]. Per il sistema redazionale questi sono i principali vantaggi derivati dall'adozione di un client web: *(i)* supporto a tutti i sistemi operativi in commercio (accesso basato su browser); *(ii)* abbattimento dei costi di distribuzione (accesso su indirizzo unico piuttosto che utilizzo di un client installato su ogni macchina).

A livello industriale i framework più utilizzati per la scrittura di SPA con Html e JS sono: Angular, React e VueJs [28]. La scelta è stata quella di adottare Angular per riutilizzare le competenze acquisite dai membri del team su altri progetti aziendali.

Nel contesto di un'applicazione web qualsiasi tipo di comunicazione deve essere gestita in modalità asincrona. Questo è dovuto alla natura single-thread di javascript: rimanere bloccati su una richiesta Http significherebbe bloccare l'esecuzione di qualsiasi altro script presente nella pagina. Quello che contraddistingue la variante implementativa della comunicazione è la direzionalità di comunicazione che vogliamo adottare, monodirezionale e bidirerezionale, entrambe necessarie al fine di produrre un'applicazione reattiva rispetto eventi utente e eventi server:

- **monodirezionale**: solitamente le richieste partono dal client web verso un server che espone servizi Rest. Queste richieste sfruttano Ajax per effettuare chiamate asincrone verso un web server. La serializzazione del dato può essere XML o JSon.
- **bidirezionale**: le richieste possono essere inizializzate da entrambi gli attori coinvolti nella comunicazione. È una delle tecniche su cui si basano i meccanismi di notifica push. Le api Html offrono i WebSocket come soluzione a questo modello di comunicazione.

Client Web	
• Descrizione	Interfaccia Web per i Redattori
• Tecnologia	HTML/Javascript con framework Angular
• Implementazione	In carico al team
Comunicazione	
• Asincrona	Ajax, Websocket

Tabella 3.1: Scheda riassuntiva del Client Web

[2] *Single Page Application*, ossia un applicazione web in cui tutti i file necessari per l'esecuzione vengono recuperati in un unico caricamento. Eventuali risorse aggiuntive sono caricate dinamicamente durante la navigazione utente

3.4.2 Api Gateway

In una architettura a microservizi, le funzionalità richieste dall'applicazione web sono spesso disseminate sui diversi servizi di backend. Per disaccoppiare il client da possibili modifiche o evoluzioni tecnologiche, viene introdotto un elemento architetturale che nasconde la topologia del backend e fornisce un'interfaccia pubblica come unico punti d'accesso da parte di consumatori esterni: l'API Gateway. Le richieste arriveranno quindi su un unico punto di ingresso, che si preoccuperà di effettuare l'indirizzamento verso il microservizio corretto. Il gateway può supportare funzionalità avanzate come il monitoraggio, throttling delle richieste e validazione dell'autenticazione.

Nel contesto del sistema redazionale, l'api gateway è supportato da due componenti infrastrutturali:

- ElasticSearch: un motore di ricerca *full-text* basato su Lucene e introdotto per soddisfare il requisito *R5*. L'Api Gateway fornisce delle chiamate Rest autenticate con la quale far interagire i vari client con tale motore di ricerca.
- SQL Azure: un database relazionale il cui compito sarà quello di immagazzinare informazioni degli utenti e i metadati dei contenuti inseriti dai redattori.

API Gateway	
• Descrizione	Canale di accesso al backend
• Tecnologia	Java con framework Spring Boot
• Implementazione	In carico al team
Persistenza	
• DB Relazionale	SQL Azure
• Ricerca Full-Text	Elasticsearch
• File system	Azure Blob Storage
Comunicazione	
• Sincrona	Servizi Rest (Http)
• Asincrona	Azure Service Bus, Websocket

Tabella 3.2: Scheda riassuntiva del servizio API Gateway

3.4.3 OCR Service

Il servizio di OCR, Optical Character Recognition, nasce dal requisito *R9* espresso dai redattori, dove dichiarano l'esigenza di poter caricare nella banca dati documenti che rappresentino del testo (es: scansioni) invece che dover riportare il contenuto a mano.

Questo servizio, è già disponibile in azienda poiché sviluppato per altre esigenze progettuali. Il pacchetto è fornito dal team come immagine Docker e sarà compito dell'utilizzatore quello di installarlo e configurarlo nel proprio ambiente di produzione.

OCR Service	
• Descrizione	Servizio di estrazione del testo da immagini
• Tecnologia	Apache Tika Server con Tesseract 4
• Implementazione	Fornita da terze parti, necessita di installazione
Comunicazione	
• Sincrona	Servizi Rest (Http)
• Asincrona	Azure Service Bus

Tabella 3.3: Scheda riassuntiva del servizio OCR

3.4.4 NLP Service

NLP, ovvero Natural Language Processing, è un ramo dell'Intelligenza Artificiale che si occupa dell'interazione tra gli elaboratori ed i linguaggi naturali. Il nostro servizio nasce infatti con lo scopo di riconoscere particolari entità all'interno di un generico testo.

In particolare, dati i requisiti R10 e R11, verrà addestrato per:

1. riconoscere riferimenti normativi (per effettuare collegamenti verso altri documenti) e persone, organizzazioni e indirizzi postali/email).
2. riconoscere anagrafiche di persone: individuazione di nomi, professioni, codici fiscali, indirizzi email, eccetera.

Nlp Service fornirà inoltre una funzionalità per l'anonimizzazione delle entità riconosciute.

La nostra implementazione sarà effettuata in Java EE, basandoci sulla libreria open-nlp di Apache[3]. Il servizio è completamente stateless e interrogabile sia tramite un'interfaccia pubblica Rest sia tramite comandi impartiti su un *topic* del servizio di messaggistica asincrono.

NLP Service	
• Descrizione	Estrazione entità da un documento testuale
• Tecnologia	Java con framework Spring Boot e Apache OpenNlP
• Implementazione	In carico al team
Comunicazione	
• Sincrona	Servizi Rest (Http)
• Asincrona	Azure Service Bus

Tabella 3.4: Scheda riassuntiva del servizio NLP

3.4.5 Document Categorization Service

Questo servizio nasce dai requisiti R3, R6 e si occupa della classificazione di un documento attraverso la tecnica di *Document Clustering*. In particolare la classificazione dei

[3] `https://opennlp.apache.org/`

documenti permette di capire sia quali sono gli argomenti trattati dal testo, sia permette di effettuare suggerimenti automatici su quali prodotti (del sistema redazionale) inserire il contenuto.

L'implementazione è già stata sviluppata dal team di Ricerca e Sviluppo nell'ambito di un progetto di ricerca finanziato. Sarà possibile utilizzare il servizio istanziando il container docker fornito dagli sviluppatori. Una volta avviato, il servizio espone un'interfaccia pubblica Rest. È possibile abilitare la comunicazione asincrona attraverso apposite configurazioni di lancio.

Document Categorization	
• Descrizione	Classificazione tramite Document Clustering
• Tecnologia	Java con framework Spring Boot
• Implementazione	Fornita da azienda, necessita di integrazione
Comunicazione	
• Sincrona	Servizi Rest (Http)
• Asincrona	Opzionale, su Azure Service Bus

Tabella 3.5: Scheda riassuntiva del servizio Document Cat

3.4.6 Pdf2Html Service

Questo microservizio nasce dalla necessità sia di poter arricchire i contenuti con riferimenti e link ad altri contenuti (*R10*), sia dalla necessità di modificare gli stessi in caso di modifiche tipo l'anonimizzazione (*R11*). Il compito del servizio consiste nel produrre una rappresentazione HTML fedele (contenuto, impaginazione, eccetera) del documento pdf in esame. La rappresentazione, dopo eventuali arricchimenti, sarà consumata direttamente dal client web e presentata in lettura all'utente finale.

Questo servizio, è già disponibile in azienda poiché sviluppato per altre esigenze progettuali. L'implementazione consiste in un'applicazione Python che incapsula la logica della libreria C *pdf2htmlEX*[4]. Il pacchetto è fornito dal team come immagine Docker e sarà compito dell'utilizzatore quello di installarlo e configurarlo nel proprio ambiente di produzione.

Pdf2Html Service	
• Descrizione	Converte un PDF nella sua rappresentazione HTML
• Tecnologia	Python, libreria C "pdf2htmlEX"
• Implementazione	Fornita da azienda, necessita di integrazione
Comunicazione	
• Sincrona	Servizi Rest (Http)
• Asincrona	Azure Service Bus

Tabella 3.6: Scheda riassuntiva del servizio Pdf2Html

[4] `https://github.com/pdf2htmlEX/pdf2htmlEX`

3.4.7 IDP Service

L'IDP, IdentityProvider, è un sistema in grado di creare, mantenere e gestire le informazioni riguardo gli utenti che accedono al nostro sistema. Uno degli scopi dell'IDP è quello di gestire l'autenticazione, verificando l'identità dell'entità che sta cercando di accedere ad una risorsa protetta.

In una scenario a microservizi, l'autenticazione viene spesso centralizzata in un singolo servizio denominato *Authentication Service*. Nel nostro caso sarà compito dell'API Gateway quello di controllare che i client abbiano un token valido per accedere ai nostri servizi. In caso negativo, inoltrerà i client verso il servizio di piattaforma. Invece in caso positivo la chiamata avrà successo e sarà inoltrata verso il microservizio opportuno, eventualmente arricchita con le informazioni dell'utente. Usando questo approccio è necessario assicurarsi che i singoli microservizi non possano essere raggiunti direttamente (senza il gateway API). In azienda è fornito come servizio di piattaforma, e quindi non sarà compito del team quello di sviluppare e mantenere un Authorization Service. Rimane tuttavia l'onere di integrarlo all'interno dell'applicativo.

IDP Service	
• Descrizione	Servizio di autenticazione aziendale
• Tecnologia	Keycloak Server
• Implementazione	Servizio di piattaforma Maggioli: idp.maggiolicloud.it

Tabella 3.7: Scheda riassuntiva del servizio IDP

3.5 Architettura finale

Nello schema evitiamo di rappresentare servizi di piattaforma forniti dall'azienda di cui non dovremo gestire direttamente il ciclo di vita: ad esempio il servizio di IdP aziendale.

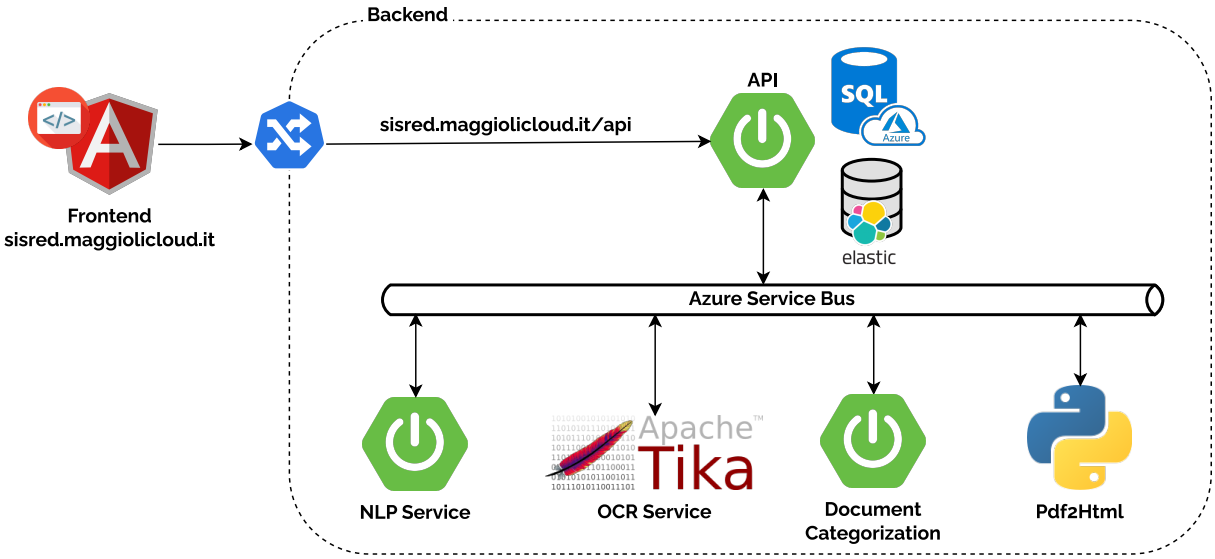

Figura 3.1: Nuova Architettura del Sisred

Capitolo 4

Ridefinizione del processo di sviluppo

4.1 Introduzione

Ora che abbiamo presentato la totalità delle componenti emerse dal rinnovamento architetturale e tecnologico del sistema redazionale, riprendiamo le pratiche emerse dallo studio dello stato dell'arte DevOps. In particolare nella rivisitazione dei processi si dovranno catturare sia quelle che sono le esigenze del team, già emerse nell'intervista riguardo l'organizzazione del sisred originale in Sezione 2.3, sia le necessità introdotte dal nuovo paradigma architetturale a microservizi visto nel Capitolo 3.

4.2 Impostazione del flusso di lavoro

Il team sarà organizzato in modo che ogni sviluppatore sia in grado di operare trasversalmente su tutti i microservizi del nuovo ecosistema. Applicato ciò, si terrà comunque conto dell'esperienza individuale nel momento dell'assegnazione di eventuali nuove funzionalità o risoluzione di problematiche.

4.2.1 Gestione del progetto

Il lavoro sarà organizzato sfruttando le pratiche proposte dall'approccio Agile ed in particolare:

- Sprint delle durata di 2 settimane.
- Stand-up meeting ogni 2 giorni.
- Incontro con le figure dei product owner al termine di ogni sprint (i due redattori di riferimento).

All'interno di uno sprint è lasciato agli sviluppatori il compito di auto-assegnarsi e/o collaborare per lo svolgimento di una determinata attività. Ad ogni microservizio sarà assegnato uno sviluppatore di riferimento con il compito di effettuare *code-review* su tutto il codice sviluppato.

Per tenere traccia e mantenere il backlog delle richieste, funzionalità e bug, lo scrum master necessita di strumenti che lo aiutino a tenere traccia del progresso, organizzare gli sprint e programmare le milestone di progetto. In questo caso il focus è stato su due strumenti: il primo deriva dalla piattaforma di versionamento aziendale, Gitlab. Il secondo è stato cercato in uno strumento con un piano gratuito, chiamato ClickUp [1].

Clickup è uno strumento che nasce per l'Agile Project Management e nel suo piano gratuito permette di gestire un team di massimo 10 persone dando come strumenti Dashboard personalizzate, Card Board e assegnazione del lavoro con date, tempi, priorità e relazioni di precedenza. Purtroppo funzionalità più avanzate come visualizzazione del Gantt e Pert non sono incluse nella versione base. Inoltre al momento della scrittura non è possibile effettuare un integrazione tra la piattaforma di versionamento e Clickup in modo da avere funzionalità automatiche basate sul lavoro degli sviluppatori; ad esempio tenere traccia del completamento di una funzionalità analizzando il messaggio di commit, oppure tenere traccia di quali commit stanno contribuendo a quale attività su un determinato Sprint.

Per questo motivo la scelta è ricaduta sulla funzionalità integrate e offerte da Gitlab, funzionalità che permettono di avere tutti gli strumenti del piano gratuito di Clickup, integrando gli automatismi basati sui commit degli sviluppatori e mantenendo la comodità di avere una gestione centralizzata di codice e pianificazione.

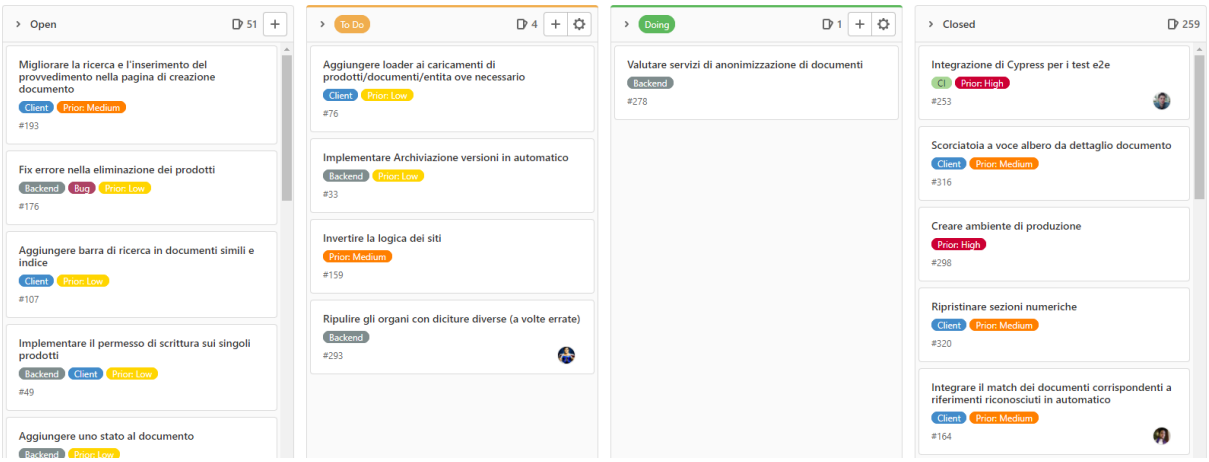

Figura 4.1: Gitlab Boards Management

4.2.2 Comunicazione

I componenti del team, seguendo anche le direttive promosse in risposta alla pandemia COVID-19, necessitavano di uno strumento che potesse agevolare la comunicazione durante il lavoro quotidiano in qualsiasi forma: scritta, orale, persistita e remota. Per tale scopo si adotta Slack[2] che grazie agli innumerevoli plugin e la possibilità di programmare da remoto la scrittura su vari canali tematici, abilita forme di automazione per centralizzare (ed eventualmente pilotare via chat) la raccolta di informazioni da sistemi tra loro eterogenei. Un primo esempio per il sistema redazionale è la gestione di un canale che raccoglie e visualizza informazioni riguardanti lo stato del repository dei sorgenti: segna-

[1] `https://clickup.com/`
[2] `https://slack.com/`

lazione di nuove problematiche, richieste di integrazioni, notifica di commit e di errori sulla pipeline automatica.

4.2.3 Ambienti

Durante lo sviluppo del nuovo sistema redazionale sarà fondamentale fornire un feedback continuo agli stackeholder interessati. Un ambiente non è altro che un installazione completa, integrata e indipendente di tutto il sistema sviluppato. A seconda dello scopo per cui nasce, nel contesto del sistema redazionale, formalizziamo i seguenti ambienti:

- **locale**: è l'ambiente rappresentato dalla macchina di sviluppo di ogni membro del team. Qui sopra si scrivono e testano le funzionalità prima ancora di pubblicarle sul repository di controllo di versione. La progettazione di questo ambiente deve trovare il giusto compromesso tra fedeltà rispetto produzione e complessità di gestione (installazione, manutenzione, performance, eccetera). Cardinalità: 1 per ogni sviluppatore.
- **review**: è l'ambiente creato appositamente per testare funzionalità che dipendono dall'integrazione di diversi microservizi. I consumatori di questo ambiente sono per lo più sviluppatori che necessitano di una *sandbox* in cui validare il risultato finale. Nel caso di modifiche complesse e prolungate nel tempo, anche project manager e product-owner hanno accesso per testing e validazione. Questa tipologia di ambienti è dinamica, ossia cessano di esistere nel momento in cui si concludere il testing. Cardinalità: 1 per ogni macro-funzionalità.
- **test**: detto anche stage o pre-produzione. È l'ambiente in cui project manager e product owner possono testare e validare le nuove funzionalità prima che raggiungano produzione. Sarà fondamentale progettare questo ambiente in modo che sia il più possibile identico a quello di produzione. Cardinalità: 1.
- **produzione**: l'ambiente reale in cui i clienti finali possono accedere ed utilizzare i servizi offerti dal sistema redazionale. Cardinalità: 1.

Vedremo nei prossimi capitoli la progettazione dell'automazione relativa ad ognuno degli ambienti in merito a gestione infrastruttura, test, rilasci e monitoraggio.

4.3 Controllo di versione

Per quanto riguarda il controllo di versione del codice i vincoli tecnologici della Sezione 2.5.2, giustificati da un omogenea adozione aziendale, impongono l'utilizzo di Git (e in particolare la piattaforma Gitlab privata Maggioli).

4.3.1 Modello di Branching

Il modello di branching consiste in una serie di politiche e convenzioni associate all'utilizzo dei rami di sviluppo di git. Di fatto quello più utilizzato in industria e letteratura è Git Flow. Questo modello si è talmente radicato nella cultura IT che le operazioni chiave associate ai vari scenari di utilizzo sono stati materializzati nelle primitive dello stesso Git.

I concetti di Git Flow:

- develop e master branch: Invece di un unico ramo per tutti i sorgenti (master), questo flusso di lavoro utilizza due rami per registrare l'avanzamento di versione del progetto. Il ramo *master* memorizza la cronologia delle versioni di produzione mentre quello di *develop* funge da ramo di integrazione per le funzionalità in sviluppo. È anche conveniente contrassegnare tutti i commit nel ramo master con un numero di versione, utilizzando i git tag.
- feature branches: ogni nuova funzionalità deve risiedere in un proprio ramo che deriva da develop e deve essere pubblicata (push) al repository centrale in modo da abilitare la cooperazione tra i vari sviluppatori. Il completamento di una feature si manifesta sotto forma di una *merge* dal ramo della funzionalità a quello di *develop*
- release branches: sono rami di appoggio che partono da *develop* quando si sono accumulate abbastanza funzionalità per giustificare un nuovo rilascio. Questi rami servono a isolare la fase di rilascio e permettono ad altri membri del team di continuare a integrare su *develop* senza rischiare di includere modifiche non volute nel ciclo di rilascio. Una volta completata la preparazione la *release* branch viene sia in *master* sia in *develop*.
- hotfix branches: queste branch sono utilizzate per applicare rapidamente *patch* alle versioni di produzione. Si effettua il branch a partire dal ramo master, si effettua l'aggiornamento, si incrementa la versione dell'applicazione e si riportano i nuovi sorgenti sia in master sia nel ramo develop.

A partire da questo standard progettiamo ora una variante che possa aderire e valorizzare le impostazioni del flusso di lavoro specificate nel capitolo precedente. Decidiamo di adottare un modello che possa permettere di avere una corrispondenza diretta tra gli ambienti richiesti e lo stato del codice:

- master branch → ambiente produzione.
- feature branch → ambiente review: mantengono la stessa semantica di git-flow, la creazione di un ambiente di review che materializza questa branch sarà automatizzato e opzionale in base all'entità della funzionalità da sviluppare.
- release branch: scompare in favore del ramo di test.
- test branch → ambiente test: è un ramo fisso che assume lo stesso ruolo del ramo release in git flow. Nel nostro modello di branching questa branch contiene il controllo di versione associato all'ambiente di pre-produzione, ovvero test. Esattamente come nelle release branch di git flow, una volta pronti per rilasciare una serie di modifiche in produzione se effettua il merge tra release (nel nostro caso test) e master.
- hotfix e develop branch: mantenute con la stessa semantica di git flow.

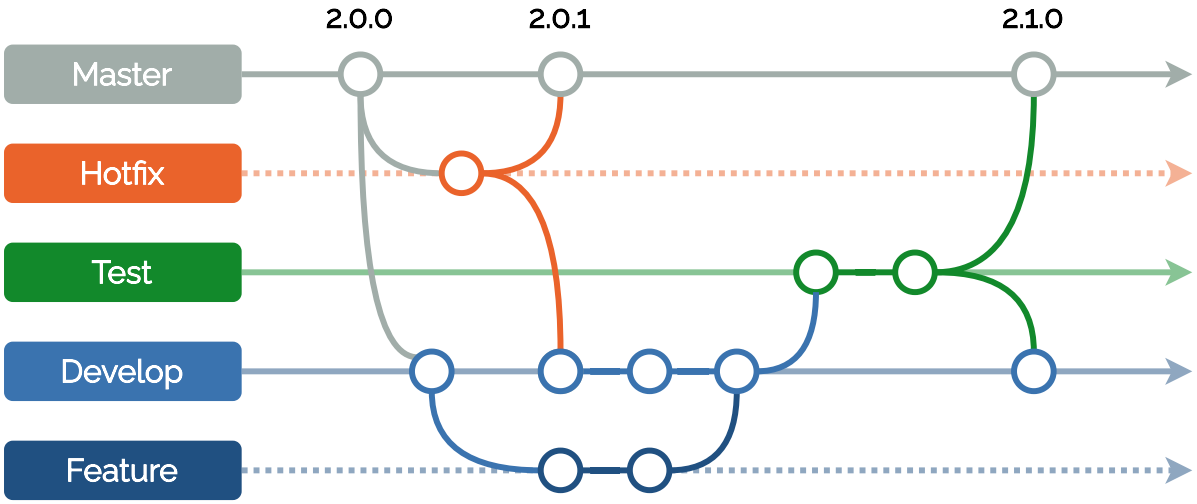

Figura 4.2: Modello di Branching adottato

Operativamente la codifica di ogni attività pianificata avviene sulle *feature/* branch. Una volta completata il programmatore richiede l'integrazione della funzionalità sul ramo *develop*. La richiesta avviene tramite una funzionalità di Gitlab chiamata *Merge Request*, l'equivalente di una Pull Request delle principali piattaforme pubbliche. Prima di accettare le modifiche, un incaricato del team effettuerà un controllo formale del codice prodotto (fase di code review) e si assicurerà che i test automatici associati alla richiesta siano terminati con successo.

Quando un numero sufficiente di funzionalità giunge sul ramo develop, allora si può procedere a integrare verso gli ambienti di stage e produzione. Il codice sul ramo di sviluppo, ha già subito delle analisi di qualità, perciò può proseguire verso test e, successivamente, master con semplici *git merge*.

4.3.2 Organizzazione del repository

Una delle scelte più importanti effettuate dal team riguarda se gestire la base di codice di ogni microservizio in un unico repository di versionamento oppure se separarli in repository diversi. Questi modelli di versionamento prendono il nome relativamente di **mono-repository** e **multi-repository**. Entrambe le soluzioni hanno vantaggi e svantaggi bilanciabili attraverso una serie di compromessi [18].

Il sistema redazionale non fa eccezione, infatti la prima soluzione da considerare è quella di seguire il principio di indipendenza del singolo microservizio e suddividerli in repository separati. Quest'approccio, seppur disaccoppi totalmente la gestione dei microservizi, introduce degli aspetti negativi amplificati dal fatto di lavorare in un team particolarmente ristretto. Ad esempio la modifica di un servizio applicativo potrebbe impattare la coordinazione di più microservizi. In questo caso per lo sviluppatore si incrementa in modo considerevole il tempo dedicato alla gestione vera e propria del repository (commit su repository diversi, cicli di rilascio differenti, assegnazione e chiusura di issue separate, code review e analisi risultano frammentate sul portale della piattaforma Git).

Dall'altro lato, utilizzare un approccio a mono-repository, significa in parte invalidare il principio di isolamento e indipendenza di un microservizio e complicare la logica di ge-

stione interna dei processi (ad esempio una pipeline automatica di rilascio dovrà tenere presente le casistiche di tutti i microservizi invece di uno solo). Tuttavia i vantaggi si traducono in facilità nell'applicare tecniche di controllo di qualità rispetto all'integrazione dell'intero prodotto. Ad esempio vedremo come creare, con un'unica pipeline, un nuovo ambiente di produzione, rilasciare sopra tutti i microservizi e validare tramite test automatici la qualità dell'integrazione. Inoltre, per gruppi ristretti come quello del sisred, è molto probabile avere sviluppatori che operano trasversalmente sulla base di codice di tutti i servizi; risulta quindi comodo avere un unico punto di controllo di versione piuttosto che orchestrarne tanti in modo separato.

Un'ultima possibilità consiste nell'attuare una soluzione ibrida: ovvero un repository per microservizio e un *meta* repository di aggregazione che li importi tutti con la tecnica del *git module*: una funzionalità che permette a un repository di essere incluso in un altro, esprimendo così il concetto di relazione di dipendenza [1]. Questa soluzione, seppur presenti grandi potenzialità, per ora non è stata presa in considerazione in quanto complica notevolmente la gestione della piattaforma di versionamento ed in particolare la configurazione delle pipeline di integrazione continua.

Il team, per il momento, decide quindi di adottare un'organizzazione a mono-repository.

4.4 Continuous Integration

L'integrazione continua è un pratica fondamentale dell'approccio DevOps che apporta numerosi vantaggi (vedi Sezione 1.5.2). Come flusso di lavoro abbiamo già stabilito la semantica di ogni ramo git e come il codice li attraversa a partire dalla *feature branch* fino a *master*.

Il successo di questa pratica non è solo determinato dalla frequenza con la quale si effettuano le integrazioni dei vari sviluppi, ma bensì determinato dalla serie di attività di controllo che vengono scatenate ogni qualvolta si recepiscono nuovi commit. Per questo motivo identifichiamo queste fasi che comporranno la fase di integrazione del sisred e che verranno scatenate su ogni evento relativo alla pubblicazione di sorgenti nel repository git (commit, merge-request, tag): *(i)* **compilazione**, *(ii)* **test**, *(iii)* **analisi**.

Per automatizzarle su una piattaforma di integrazione, lo strumento che occorre introdurre è quello del *build automator* (automazione dello sviluppo), ovvero un software spesso specializzato su uno specifico linguaggio o framework, in grado di gestire il processo di compilazione dei sorgenti nei relativi pacchetti finali. Non esiste uno strumento generico che possa gestire tutti i sistemi anche se spesso si compongono di funzionalità comuni:

- Gestione delle dipendenze
- Compilazione e linking
- Gestione del ciclo di vita del codice auto-generato (generazione, inclusione, config)
- Controllo di qualità del codice sorgente
- Gestione dei test (lancio, reportistica, copertura)
- Generazione della documentazione
- Generazione del pacchetto di rilascio
- Distribuzione del pacchetto

Per limitare la complessità di gestione, quando possibile, identifichiamo un unico strumento di automazione dello sviluppo per tipo di tecnologia piuttosto che per singolo microservizio.

	Build Automator	**Configuratore**	**Richiede**
Java	Maven	pom.xml	Jdk
HTML/JS	Yarn per gestione dipendenze, NgCli per compilazione e pacchettizzazione	package.json, angular.json	NodeJS
Python	Pip per gestione dipendenze	requirements.txt	Python

Tabella 4.1: Scheda riassuntiva degli strumenti di Automazione dello sviluppo

La conoscenza delle dipendenze di ogni build automator è fondamentale per progettare l'ambiente in cui dovrà operare la piattaforma di automazione (cioè la piattaforma che implementerà l'intera catena di CI/CD).

Di seguito analizziamo i passi individuati per l'integrazione continua, avendo cura di riportare eventuali considerazioni rispetto i rami del nostro flusso di lavoro.

4.4.1 Compilazione

L'obiettivo di questo passo è quello di capire se i sorgenti di un determinato ramo git compilano correttamente. In genere prima di effettuare la compilazione, si procede con il recupero delle dipendenze dichiarate dal progetto.

Per quanto riguarda **Maven**, essendo basato sul concetto di *build lifecycle*, non è necessario esplicitare alcuna azione di download; ogni comando predefinito appartiene ad una lista associata a momenti ben definiti del ciclo di vita del codice. Lanciare un comando significa indirettamente lanciare anche quelli che lo precedono.

Lo stesso discorso non si può fare per lo strumento associato al client Web. Nel framework Angular infatti, la gestione delle dipendenze è disaccoppiata dal sistema in grado di effettuare la build. La fase di download è esplicita e la dichiarazione è standardizzata in un file denominato *package.json*. I principali vendor in grado di adempiere al compito del package manager sono tre:

- **Npm**: package manager storico, inizialmente rilasciato insieme a NodeJs.
- **Yarn**: rispetto a npm introduce una cache e download parallelo dei pacchetti.
- **Pnpm**: come Yarn, ma introduce una serie di ottimizzazioni nella gestione del numero di pacchetti scaricati. Se 100 pacchetti dipendono dalla libreria *loadash*, pnpm scaricherà un solo loadash e verrà referenziato da tutti i 100 pacchetti tramite symbolic link. In caso di gestione di versioni diverse vi è anche un'ottimizzazione tramite *delta* dei soli file modificati.

Sulla base di test empirici lo strumento più performante è Pnpm[3]; si decide tuttavia di utilizzare Yarn, sia per riutilizzare l'esperienza fatta in progetti precedenti, sia per il livello

[3] `https://archive.vn/FnnwC`

di prestazione simile in presenza di cache e node_modules (requisito da imporre nell'ottimizzazione del rilascio automatico). Inoltre, seppur il client web si basi su linguaggi di scripting interpretati come Javascript, il framework Angular ci permette di scrivere codice in Typescript, ovvero un linguaggio in grado di essere compilato in Javascript nativo e quindi interpretabile dal browser. Sarà questo, oltre a numerose fasi di ottimizzazione, il compito assegnato alla Angular Cli (che non fa altro che creare uno strato opinionato di configurazioni al famoso strumento di impacchettamento dei moduli web (letteralmente web module bundler): *Webpack*[4].

4.4.2 Test

Questa fase succede quella di compilazione e si basa sugli artefatti prodotti da essa. L'obiettivo è quello di verificare la correttezza del comportamento di ogni microservizio. Nella scelta del giusto framework di testing abbiamo considerato le seguenti caratteristiche chiave: la facilità di integrazione con il nostro strumento di automazione dello sviluppo e la possibilità di esportare report (un sommario HTML/Json di esecuzione con relativo esito o un analisi della copertura dei test sulla base di codice) del risultato dell'esecuzione dei test, eventualmente consumabile dalla piattaforma di CI/CD.

Durante la fase di integrazione continua le due tipologie di test che vogliamo includere sono i test unitari e quelli di integrazione. Compatibilmente con i requisiti sopra esposti individuiamo i seguenti strumenti.

- **Test unitari**: JUnit5 per microservizi Java, *Jest* per client web in Angular. Lo scopo è testare la singola funzionalità, isolandola dal contesto.
- **Test di integrazione**: Spring Test Context framework per microservizi Java Spring Boot. Attraverso il contesto di Spring è possibile effettuare test su macro funzionalità, ad esempio testare se l'invocazione di una rest api che inserisce un documento provoca l'inserimento di un evento sul bus di messaggistica.

4.4.3 Analisi statica

Come già visto in Sezione 1.5.4, nel contesto dell'integrazione continua, l'analisi statica è il processo di valutazione di un sistema o di un suo componente basato sulla sua forma, sulla sua struttura, sul suo contenuto o sulla documentazione di riferimento. Ciò significa che la valutazione avviene a prescindere dall'esecuzione del sistema o dell'oggetto che si sta testando.

Questa fase della CI succede quella di testing e viene presa in considerazione solo se i passi precedenti terminano con successo. Questo perché non si vuole *investire* tempo e risorse nell'analisi di codice che non verrà mai mandato in produzione.

In questo contesto adottiamo lo strumento **Sonarqube**, sia per la grande diffusione all'interno del settore IT, sia perché a livello aziendale è presente un'installazione di Sonarqube Server Community sulla quale sarà possibile integrare i propri test: `sonarq.maggioli.it`. L'analisi statica del codice con Sonarqube consiste in due componenti: uno *scanner* specifico per il linguaggio da analizzare e un server che gestisce la fase di analisi e reportistica. Durante l'analisi lo scanner collezionerà tutti i dati necessari; il server invece effettuerà le

[4] `https://webpack.js.org/`

valutazioni necessarie tenendo presente sia delle configurazioni passate dal progetto che richiedo lo scan e delle *regole* associate ad un certo linguaggio. Queste regole, quando violate, producono una issue, ovvero una problematica. Appartengono a diversi domini:

- Code Smell (dominio della manutenibilità): tecnicamente non sono errori bloccati ma indicano potenziali debolezze nella progettazione
- Bug (dominio dell'affidabilità)
- Vulnerabilità (dominio della sicurezza): problemi di sicurezza nel codice
- Hotspot di sicurezza (dominio della sicurezza): sono frammenti di codice segnalati perché potenzialmente insicuri. Tuttavia il contesto applicativo potrebbe renderlo accettabile. Necessitano di code review manuale.

Tramite queste definizioni, SonarQube fornisce il concetto di **Quality Gate**, ovvero un giudizio rilasciato dallo strumento e basato su una serie di indicatori, che riporta lo stato attuale dell'analisi: *approvato* o *rifiutato* [9].

Nel contesto del sistema redazionale configuriamo SonarQube per rifiutare in caso di una delle seguenti: *(i)* un bug con severità bloccante; *(ii)* una vulnerabilità di sicurezza. Successivamente sfrutteremo lo stato di questo quality gate per interrompere o meno l'esecuzione della pipeline automatica di CI/CD.

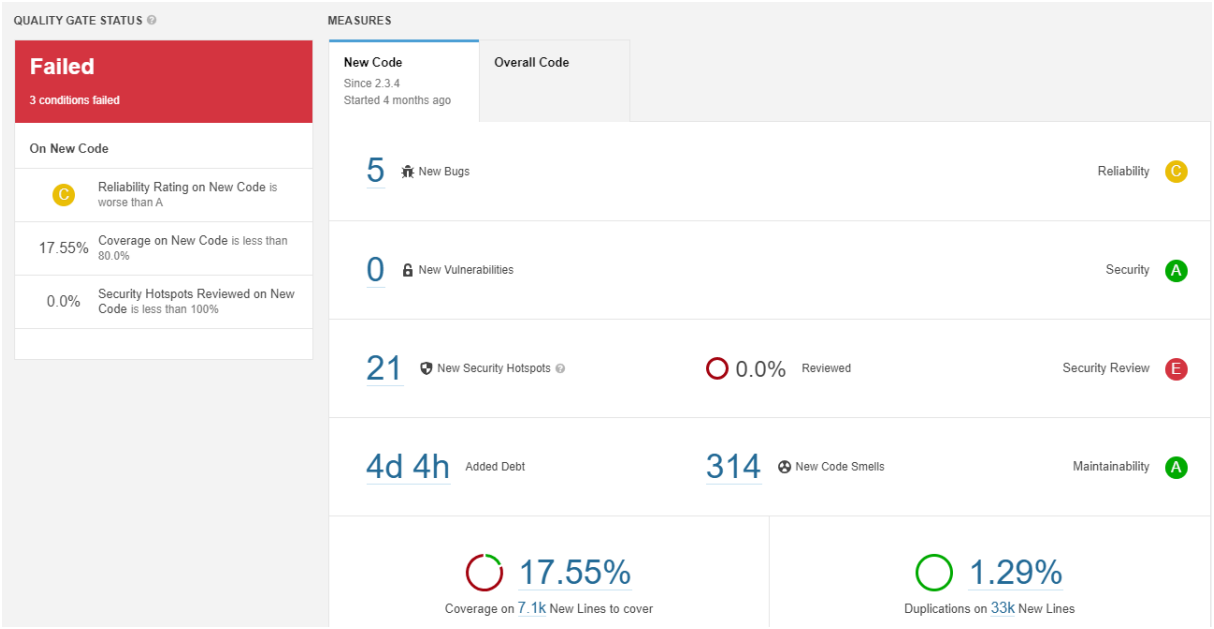

Figura 4.3: Sonarqube quality gate nella dashboard di progetto

Per tutti i linguaggi utilizzati dai microservizi del sistema redazionale è previsto un plugin, integrabile nello strumento di automazione dello sviluppo, in grado di lanciare l'analisi statica con Sonarqube. In Maven, ad esempio, includeremo il sonar-maven-plugin fornito proprio dagli stessi sviluppatori di Sonar. Per il progetto web è presente invece un pacchetto sul registro di npm chiamato sonarqube-scanner, basterà includerlo nel package.json e richiamarlo al momento opportuno durante la pipeline. Al momento nella versione community non è possibile differenziale tra branch differenti dello stesso progetto; per questo motivo scegliamo di scatenare l'analisi statica solo per i sorgenti presenti sul ramo develop.

4.5 Continuous Testing

Nel precedente capitolo abbiamo visto come e quando effettuare testing durante CI. Sappiamo però che DevOps promuove una metodologia di testing continuo, che non si limita ad un singolo momento temporale della fase di rilascio. Vedremo quindi come estendere tale ciclo di rilascio, che già considera il livello di integrazione continua, con altri due livelli.

4.5.1 Livello locale

Sfruttando i vantaggi derivati dall'utilizzo di framework moderni, possiamo integrare nel flusso di sviluppo locale l'utilizzo della tecnica del **live-reload**. Il live-reload è una tecnica, spesso offerta dallo strumento che gestisce il framework, che ha come obiettivo quello di monitorare la base di codice in attesa di modifiche da parte dello sviluppatore. Una volta individuate sarà compito suo quello di processarle con il giusto compilatore e, successivamente, presentare la versione aggiornata dell'applicazione allo sviluppatore (eventualmente a caldo)[5] nell'ambiente locale in cui sta avvenendo lo sviluppo.

Questa pratica non solo velocizza il flusso di sviluppo, ma consente anche allo sviluppatore di testare (sia come compilazione, sia come comportamento) prontamente le modifiche ancora prima di entrare nel ciclo di pipeline. Nel caso del sisred sia Spring Boot con Maven sia Angular con l'ng-cli forniscono supporto diretto al live-reload.

4.5.2 Livello post rilascio

Questo livello nasce dalla necessità di dover validare automaticamente la qualità dell'applicazione rilasciata, nella sua interezza. Di conseguenza a livello di ciclo di rilascio del software si porrà come ultimo tassello, sequenzialmente dopo il rilascio dell'infrastruttura e dell'applicativo.

Per tale scopo ci avvaliamo di una metodologia di testing chiamata **End To End** (anche conosciuti come E2E), che consiste nel simulare l'interazione di un utente con l'applicazione con l'intento di *stimolare* vari scenari di utilizzo. Anche se la simulazione avviene dalla prospettiva dell'utente finale, le funzionalità *stimolate* possono avere effetti su tutti i livelli dello stack applicativo [5]. Lo strumento che permette di scrivere e lanciare questi E2E test deve anche permettere di effettuare introspezione dell'interazione e raccolta dei risultati. Nel corso della progettazione sono stati analizzati due possibili strumenti: Protractor e Cypress.

Il primo, **Protractor**, nasce per l'integrazione con applicazione scritte in Angular. Le sue api incapsulano e si basano su quelle esposte da Selenium e WebDriverJS. Una delle funzionalità più interessanti è il Cross-Browser testing, che permette di lanciare parallelamente la stesso test su diversi tipi di browser. Durante l'utilizzo il difetto principale riscontrato è stato quello della difficoltà di debug: quando un test falliva non era subito chiaro quali fossero le cause scatenanti: effettivo bug? problema di latency? problema nei test?

[5] si intende senza necessità di spegnere e riaccendere alcun ambiente

Cypress invece si propone come il testing framework di nuova generazione. Relativamente giovane come sviluppi e indipendente dalle api già note in industria: non è infatti basato su Selenium Web Driver. Rispetto ai framework già collaudati, Cypress offre delle funzionalità che risolvono i problemi riscontrati utilizzando Protractor:

1. Consistenza dei risultati: il framework prevede di ritentare o aspettare in automatico quando elementi del DOM previsti non sono ancora presenti in pagina. Questo consente di evitare inutili fallimenti quando si ha a che fare con diverse condizioni di rete non conoscibili a priori.
2. Test degli scenari limite: possibilità di sviluppare test includendo scenari di latenza, errori di rete, eccetera.
3. Funzionalità *Viaggio nel Tempo*: durante l'esecuzione di un test vengono effettuati snapshot continui a cui è possibile accedere in modalità estemporanea.
4. Screenshots e Video: immagini e video completi dell'errore vengono registrati in automatico in presenza di un fallimento

Quest'ultima caratteristica è stata fondamentale per il passaggio a Cypress in quanto fornisce agli sviluppatori un strumento potentissimo per poter analizzare condizioni di errore in seguito a rilasci su ambienti di produzione reali. Durante la CI/CD basterà integrare un meccanismo che ci consenta di archiviare questi contenuti multimediali per poterli visionare in un secondo momento.

Dal punto di vista dell'automazione, questo tipo di analisi necessita di agire sulla totalità delle parti rilasciate e, per tale motivo, si colloca come ultimo passaggio dell'intera filiera di rilascio.

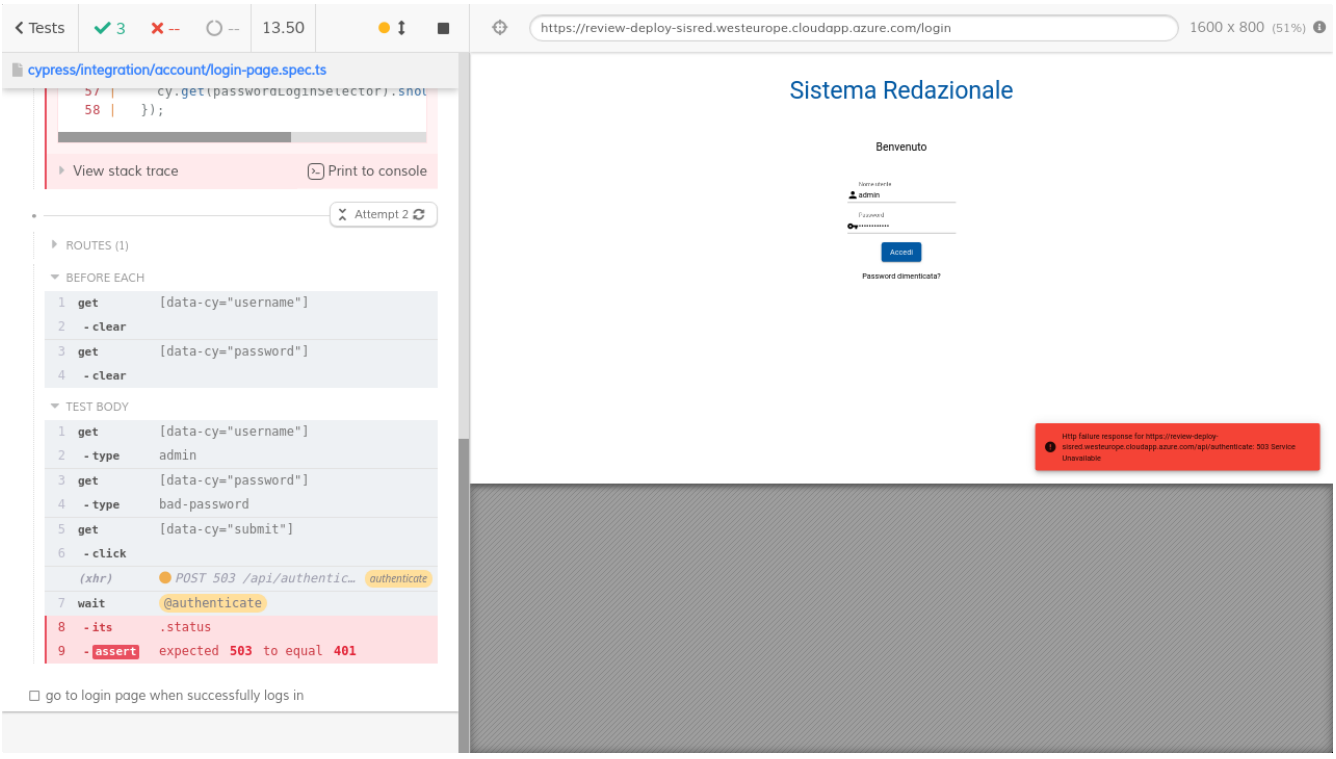

Figura 4.4: Esempio di screenshot eseguito durante e2e

4.6 Continuous Inspection

Come per la pratica del testing, anche per l'ispezione, l'approccio DevOps promuove la mentalità dell'esecuzione continua, che non si limita ad un unico punto temporale. Le ispezioni, per loro natura, spesso lavorano a livello statico analizzando il codice sorgente e quindi è comodo anticipare questa analisi direttamente nella macchina dello sviluppatore nel momento di scrittura del codice. Di seguito le diverse tipologie di analisi individuate nel contesto del sisred.

4.6.1 Analisi statica locale

Con questa pratica l'analisi statica è effettuata già durante lo sviluppo. L'obiettivo è quello di intercettare eventuali problemi ancora prima di pubblicarli sul repository di versionamento del codice. In questo ambito uno strumento molto utilizzato è SonarLint.

SonarLint è un'estensione per i principali IDE che ti aiuta a rilevare e risolvere i problemi di qualità durante la scrittura del codice. Come un controllo ortografico, SonarLint elimina i difetti in modo che possano essere risolti prima di eseguire il commit del codice. Il principale vantaggio consiste nel fatto che la problematica viene visualizzata istantaneamente sul frammento di codice che genera problemi ed il programmatore può correggere senza bisogno di analizzare un report di qualità in modo estemporaneo alla scrittura del codice stesso. Sonarlint è rilasciato come plugin disponibile per tutti i principali IDE. Viene distribuito con regole predefinite per ogni linguaggio ed è possibile sincronizzare le proprie configurazioni con quelle eventualmente definite su un server Sonarqube.

4.6.2 Regole di stile

Adottare uno stile di programmazione ben preciso può aiutare a garantire che il codice sia conforme alle buone pratiche di programmazione. Il che equivale a migliorare la qualità, leggibilità, riutilizzabilità del codice e quindi ridurre i costi di sviluppo. Questo tipo di analisi è svolto da diversi strumenti, che variano in base al linguaggio in esame e in base al momento temporale di esecuzione dell'ispezione. In questa ottica la migliore sinergia è stata individuata negli strumenti EditorConfig, Java CheckStyle e ESLint.

EditorConfig definisce semplicemente uno stile di formattazione del codice standard tra tutti gli IDE e gli editor utilizzati all'interno di un team di sviluppatori. Ad esempio, se un team utilizza due editor principali come Intellij e Visual Studio Code, EditorConfig consente loro di definire un modello di rientro comune (spazi o tabulazioni) all'interno di un singolo file. EditorConfig è ormai uno standard per tutti gli IDE moderni e per abilitarne l'utilizzo basterà introdurre il relativo file di configurazione nella root di progetto: *.editorconfig*

Checkstyle e **ESLint** sono invece strumenti orientati relativamente a Java e Javascript e permettono di analizza staticamente il codice per segnalare incongruenze rispetto allo stile di riferimento (numero massimo di caratteri su una riga, forza l'uso del ';' in javascript, a capo dopo parentesi graffa, eccetera).

4.6.3 Scansione vulnerabilità

Con vulnerabilità si intende un problema tecnico, un difetto o un punto debole presente nel software o in un sistema operativo. Anche in questo caso l'obiettivo è quello di individuare uno strumento che sia in grado di effettuare una scansione per la rilevazione di tali vulnerabilità e che possa inoltre essere pilotato da un ciclo di rilascio automatico.

Identifichiamo in **Trivy** lo strumento necessario per adempiere a tali requisiti. Da precedente analisi, Sezione 3.3, i container sono il mezzo adottato dal sisred per la distribuzione dei microservizi. Trivy permette di analizzare questi container con lo scopo di individuare vulnerabilità dei pacchetti del sistema operativo (Alpine, RHEL, CentOS, ecc.) o delle dipendenze delle applicazioni (Bundler, Composer, npm, yarn, eccetera). Lo strumento lavora con un database periodicamente aggiornato con le ultime *CVE*[6].

Dovendo analizzare i container generati di ogni microservizio, a livello di automazione questo strumento si deve scatenare solo dopo l'effettiva generazione dell'immagine docker del microservizio. Come già fatto precedentemente per altre analisi, presentiamo un report delle vulnerabilità e opzionalmente si può impostare un *quality gate* per interrompere la pipeline di rilascio in caso di rilevamento di almeno una vulnerabilità di sicurezza grave.

```
psacr.azurecr.io/sisred-api:47cca491 (alpine 3.12.3)
====================================================
Total: 2 (HIGH: 2, CRITICAL: 0)

+--------------+------------------+----------+-------------------+---------------+---------------------------------------+
|   LIBRARY    | VULNERABILITY ID | SEVERITY | INSTALLED VERSION | FIXED VERSION |                 TITLE                 |
+--------------+------------------+----------+-------------------+---------------+---------------------------------------+
| libcrypto1.1 | CVE-2021-23840   | HIGH     | 1.1.1i-r0         | 1.1.1j-r0     | openssl: integer                      |
|              |                  |          |                   |               | overflow in CipherUpdate              |
|              |                  |          |                   |               | -->avd.aquasec.com/nvd/cve-2021-23840 |
+--------------+                  +          +                   +               +                                       +
| libssl1.1    |                  |          |                   |               |                                       |
|              |                  |          |                   |               |                                       |
|              |                  |          |                   |               |                                       |
+--------------+------------------+----------+-------------------+---------------+---------------------------------------+
```

Figura 4.5: Esempio di report di vulnerabilità su microservizio api gateway

4.6.4 Aggiornamento dipendenze

Adottare un sistema di aggiornamento automatico delle dipendenze significa incrementare la qualità del software erogato per rapidità nel: *(i)* adottare nuove funzionalità e ottimizzazioni; *(ii)* recepire fix di vulnerabilità e patch di sicurezza. Nei servizi di piattaforma per il versionamento del codice pubblici, tipo *Github*, è ormai presente da diversi anni un servizio integrato di aggiornamento delle dipendenze chiamato Dependabot. Nel contesto del sistema redazionale la piattaforma è privata e questo tipologia di servizio non è presente. Per tale motivo si decide di implementare un servizio, in primis per il sistema redazionale, che sarà poi riusabile da tutti i progetti in azienda.

Renovate Bot

Renovatebot[7] è uno strumento open source in grado di effettuare la scansione di un repository git al fine di: *(i)* individuare qualsiasi file che contiene una definizione delle dipendenze; *(ii)* valutare per ciascuna dipendenza se esiste una versione più recente; *(iii)* applicare

[6]Common Vulnerabilities and Exposures (CVE)
[7]`https://github.com/renovatebot/renovate`

su un ramo separato l'aggiornamento e validarlo tramite pipeline di CI; (*iv*) integrare l'aggiornamento in caso di successo.

Nel contesto del sistema redazionale questo strumento effettua la scansione ogni ora, tramite una pipeline isolata e separata dai normali processi di CI/CD scatenati sul repository del sisred. Inoltre è stato aggiunto un passo intermedio che consiste nella presentazione di una panoramica dello stato delle dipendenze piuttosto che di un'esecuzione di aggiornamenti senza governo. In questa dashboard il programmatore può scegliere puntualmente cosa aggiornare tramite l'utilizzo di una *check*. Il successivo flusso di aggiornamento prosegue in modo automatico.

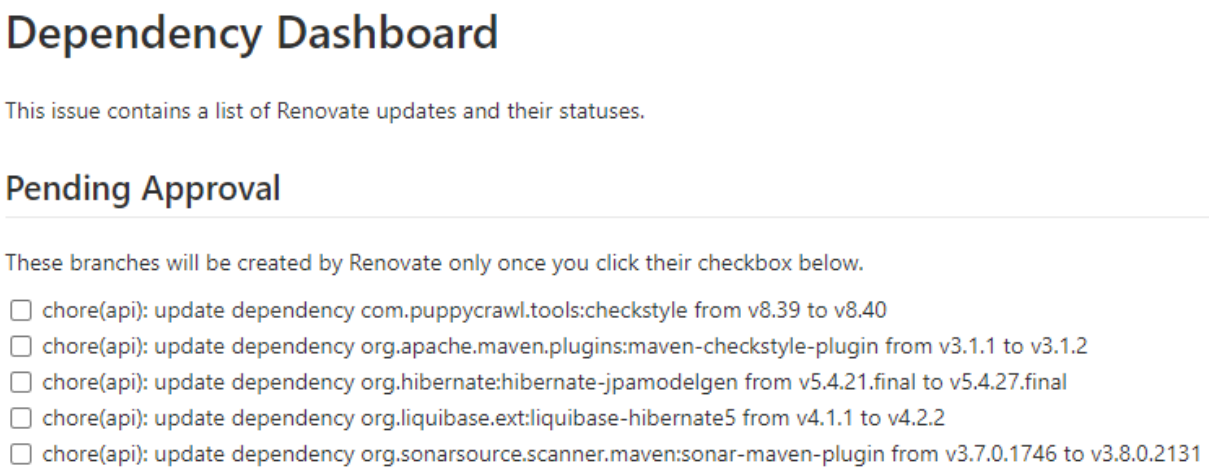

Figura 4.6: Dashboard aggiornamento dipendenze

4.7 Continuous Deployment

Le esigenze del team emerse durante l'intervista sullo stato attuale del sistema, sezione 2.3, esprimono chiaramente di voler superare i colli di bottiglia dovuti alla suddivisione del lavoro tra vari verticali, in particolare i momenti morti in cui si aspettava la creazione di un ambiente e la relativa installazione di tutto l'ambiente in grado di ospitare il sisred. Si decide quindi di adottare pienamente l'approccio continuo promosso da DevOps, implementando una logica di processi che possa supportare il rilascio continuo delle modifiche prodotte dagli sviluppatori. In particolare, seguendo la semantica legata al modello di branching, si dovranno progettare le misure per pilotare il rilascio degli ambienti di review, test e produzione.

4.7.1 Package

L'obiettivo di questa fase è quello di produrre un pacchetto software pronto per essere collocato nel relativo ambiente di produzione. Ancora una volta questo tipo di attività può essere svolto dal relativo strumento di automazione dello sviluppo, utilizzando l'adeguato profilo creato per le regole di produzione.

Per i microservizi sviluppati in Java significa produrre un archivio di tipo *jar* o *war* eseguibili e che incorporino l'eventuale logica di application server. Invece per il microservizio dedicato al client web si tratterà di produrre una cartella *dist* contenente tutti i file ottimizzati e compressi per produzione (in gergo detti anche minificati, **minified**).

```
1  FROM adoptopenjdk/openjdk14:alpine-jre
2
3  ENV SPRING_OUTPUT_ANSI_ENABLED=ALWAYS \
4      JAVA_OPTS=""
5
6  WORKDIR /app
7  COPY target/*.jar ./app.jar
8
9  CMD java ${JAVA_OPTS} -noverify -XX:+AlwaysPreTouch -jar /app/app.jar
10
11 EXPOSE 8080
```

Listato 1: Dockerfile del microservizio Api Gateway

4.7.2 Publish

Una volta generato il pacchetto pronto per essere distribuito non ci resta che pubblicarlo con il metodo di distribuzione scelto. In Sezione 3.3 abbiamo discusso il motivo dell'adozione dei container (e in particolare l'implementazione tramite Docker).

Distribuire l'applicazione tramite immagini Docker significa dover progettare, per ogni microservizio, il relativo **Dockerfile**, ovvero un documento di testo contenente tutte le istruzioni necessarie per assemblare un'immagine in grado di ospitare l'applicazione desiderata. Generati i Dockerfile, l'intero processo può essere automatizzato nella fase di CI/CD tramite il relativo comando di build che si baserà, appunto, su tali file.

```
docker build -f Dockerfile
```

Da notare nel listato 1 come il Dockerfile alla riga 7 sfrutti il pacchetto jar generato nella fase *package* del processo di CD. È possibile visualizzare una variante di questo approccio nel listato 2, dove si utilizza la cosiddetta tecnica di **multistage build** ovvero un tipo di pacchettizzazione che prevede di materializzare le diverse fasi della CI/CD all'interno del Dockerfile. Ogni fase viene eseguita in una propria *sandbox* costituita dall'immagine docker di partenza (la FROM) e può fare riferimento ad artefatti prodotti dalle fasi precedenti tramite la funzione di copia. Di seguito mostreremo la trasformazione in multi-stage dello stesso microservizio preso in considerazione nell'esempio precedente:

In questo modo, l'ultimo stage (che inizia dal *FROM* di riga 17) rappresenta l'effettiva immagine finale del processo di compilazione, ovvero quella che dovrà ospitare il pacchetto di produzione senza tutti gli strumenti necessari per effettuare compilazione, test e ispezione. Tutte le immagini precedenti sono propedeutiche alla realizzazione di artefatti o test e non saranno inclusi nel prodotto finale.

Durante la progettazione del sistema redazionale, si è sperimentato l'utilizzo di questa tecnica e di seguito riportiamo le considerazioni raccolte, suddividendole in vantaggi e svantaggi rispetto alla prospettiva del nuovo sistema redazionale. I vantaggi:

- Il processo di build, controllo qualità e creazione ambiente con relativo pacchetto di produzione sono documentati e descritti in un unico Dockerfile (per microservizio). Nessuna frammentazione della conoscenza tra codice e sistema di CI/CD.
- La quasi totalità della pipeline è materializzata nel Dockerfile e l'eventuale piattaforma di CI/CD non deve far altro che avviare il processo di build con il comando

```
FROM maven:3-adoptopenjdk-14 as builder

WORKDIR /app

COPY pom.xml .
RUN [ "mvn", "dependency:go-offline" ]

COPY ./ /app/
RUN [ "mvn", "install", "-Pprod", "-DskipTests" ]

FROM adoptopenjdk/openjdk14:alpine-jre

ENV SPRING_OUTPUT_ANSI_ENABLED=ALWAYS \
    JAVA_OPTS=""

WORKDIR /app
COPY --from=builder /app/target/*.jar ./app.jar

CMD java ${JAVA_OPTS} -noverify -XX:+AlwaysPreTouch -jar /app/app.jar

EXPOSE 8080
```

Listato 2: Dockerfile multistage del microservizio Api Gateway

docker build. Non avviene un legame forte verso una specifica piattaforma di CI/CD e la migrazione verso un nuovo sistema consisterà nel programmarlo per lanciare la build dell'immagine docker.
- Semplicità di debug dell'intera pipeline in un qualsiasi ambiente che supporti l'installazione di docker.
- *Docker build* pressoché istantanee in scenari in cui si compila la stessa immagine docker senza invalidare la cache (es: compilazioni consecutive in cui non cambiano sorgenti e librerie)

Di contro, sono stati evidenziati i seguenti svantaggi:

- Build molto lente quando non è possibile sfruttare la cache di pipeline precedenti.
- In uno scenario di CI/CD, quando viene scatenato un job è quasi sempre per aggiornamento sorgenti/dipendenze, quindi la cache nativa di Docker spesso risulta invalidata.
- Nel caso di utilizzo di una piattaforma di build che utilizzi N>1 nodi slave per la gestione della Pipeline, la cache e il repository locale delle immagini Docker non è distribuita. Questo significa che nel caso di job che sfruttano immagini compilate in job precedenti, potrebbero fallire (es: nel caso di un job dedicato al *docker build* e uno al *docker push*, il secondo potrebbe fallire poiché nel suo nodo quell'immagine non esiste).
- Nessun supporto in Gitlab per l'estrazione di artefatti dai container utilizzati per implementare il multistage (es: nel caso in cui volessimo estrarre il report di esecuzione dei test per visualizzarlo nel pannello della merge request).
- Non è possibile avere stage intermedi *opzionali*, ovvero che permettano di continuare la build anche in caso di fallimento.
- Non è possibile lanciare in parallelo stage tra loro indipendenti.

A livello operativo, l'impatto descritto nell'analisi degli svantaggi, supera di gran lunga l'apporto dato dai vantaggi. Il risultato consisteva in pipeline di rilascio molto instabili, dalla lunga durata e con artefatti non integrabili con le funzioni native date dalla nostra piattaforma di CI/CD (integrazione report, generazione pacchetto scaricabile, cache distribuita). Si è deciso quindi di rimanere su un approccio *classico* in cui il compito di Docker è quello di fornire un ambiente in cui impacchettare gli artefatti prodotti in passi precedenti del ciclo di rilascio.

Per completare il processo di **pubblicazione**, ora che sappiamo come generare la nostra immagine docker, non ci resta che renderla visibile agli utilizzatori. Con Docker, infatti, il servizio demone che pilota il processo di build, pubblicherà il risultato di tale operazione in un repository locale alla macchina utilizzata. Il passo complementare consiste nell'assegnare un **tag** (una stringa che identifica in modo univoco l'artefatto) all'immagine generata e pubblicarla su un registro in modo che possa essere visibile da altre macchine.

La figura 4.7 mostra chiaramente il nostro scenario di riferimento. Kubernetes rappresenta l'ambiente finale incaricato di *orchestrare* i container. Si noti come il ruolo della pipeline rispetto all'ambiente di produzione non sia quello della pubblicazione fisica degli artefatti, ma bensì l'applicazione di file di configurazione descrittivi (chiamati **Manifest**) che istruiscono l'orchestratore riguardo dove reperire le immagini docker precedentemente pubblicate.

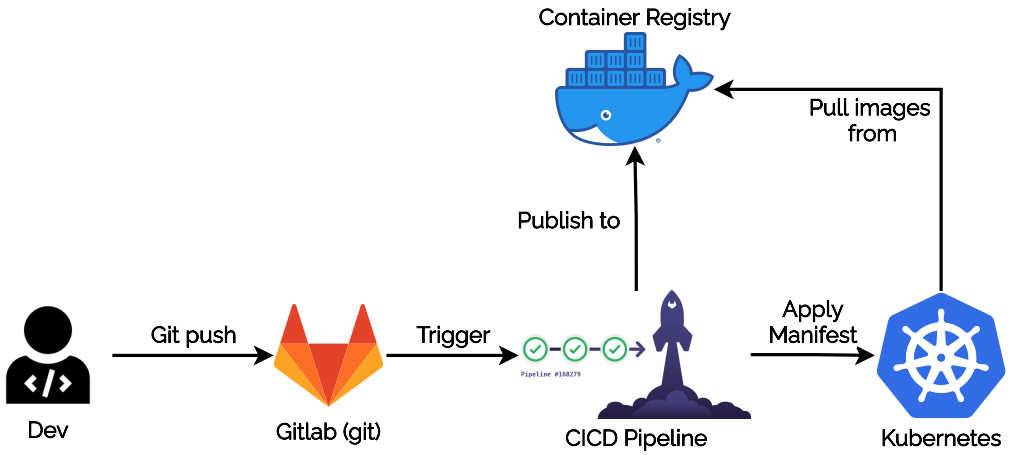

Figura 4.7: Ciclo di vita delle immagini docker

Nel processo di pubblicazione rimangono da effettuare due scelte: la politica di creazione dei tag delle immagini docker e su quale registro pubblicarle.

Registro Docker

Il registro consiste in un servizio di backend stateless e altamente scalabile che archivia e consente di distribuire le immagini Docker. Il registro è open-source e distribuito sotto la licenza permissiva di Apache. Questo abilita diversi scenari di utilizzo:

1. utilizzo del registro pubblico e gratuito di Docker Hub[8]: al momento della scrittura di questa analisi permette di gestire gratuitamente un numero ristretto di repository

[8] `https://hub.docker.com/`

privati.
2. gestione in casa di un registro privato: la gestione di backup, aggiornamento ed esposizione verso reti esterne comportano complicazioni aziendali sia dal punto di vista operativo che di quello delle politiche di sicurezza.
3. utilizzo di un servizio cloud: rappresenta la soluzione effettivamente adottata. In particolare il servizio **Azure Container Registry (ACR)**, presente nel nostro Cloud di riferimento ci permette di istanziare velocemente e a basso costo un repository con backup automatico, gestione della retention (ovvero dopo quanto tempo cancellare automaticamente un'immagine non più in uso) ed eventualmente un servizio di scansione delle vulnerabilità.

Politica di creazione delle versioni

Si tratta di stabilire la modalità di assegnazione dei nomi alle immagini dei container. L'unico requisito che si vuole rispettare è quello di non prevedere l'intervento manuale durante la pipeline. Di conseguenza le alternative valutate sono:

1. ereditare i **tag** effettuati su git: utilizzare questa soluzione permette di pilotare dei nomi di tag *human-readable* ma dall'altro lato ci pone davanti a due grandi limiti.
   (a) Il nostro workflow non prevede di effettuare tag per *scatenare* le pipeline di rilascio.
   (b) A livello organizzativo, si è stabilito di utilizzare un mono-repository per tutti i microservizi. Gestire tramite git tag il versionamento di tutte le relative immagini docker sarebbe di difficile controllo.
2. utilizzare il **timestamp** del commit: facilmente automatizzabile e con nome semanticamente intuitivo.
3. utilizzare lo **SHA** del commit di git: come il timestamp rimane facilmente automatizzabile. In questo caso perdiamo la facilità di interpretazione ma rispetto alla gestione a timestamp gli sviluppatori guadagnano la mappatura 1 a 1 tra identificativo del commit e un'immagine pubblicata (velocizzando quindi anche il processo di debug e pull in locale). Si opta quindi per l'utilizzo di questa modalità.

Quando pubblicare

La pubblicazione di un immagine docker sul relativo registro, in un contesto di continuos deployment, occorre effettuarla solo quando l'obiettivo della pipeline è quello di aggiornare uno degli ambienti previsti in Sezione 4.2.3. Questo, oltre ad evitare uno step di CI/CD *superfluo*, ci permette anche di risparmiare spazio e utilizzo di banda, i due parametri utilizzati per il calcolo dei costi del servizio cloud ACR.

4.7.3 Deploy

Nella fase di publish le immagini docker di ogni microservizio sono compilate e pubblicate sul relativo Container Registry. Ora non ci resta che studiare una strategia per applicarle in produzione. Dall'analisi precedente sappiamo che esistono due tipologie di ambienti: una tipologia statica istanziata manualmente (test e master) e una dinamica (review-*) ovvero istanziata automaticamente quando necessario.

Entrambe le tipologie di ambiente prevedono, come giustificato in Sezione 3.3, di adottare un orchestratore di container, in particolare Kubernetes (abbreviato k8s). Come già visto

in precedenza alla base del funzionamento di kubernetes vi è l'utilizzo di un **approccio dichiarativo** con il quale possiamo descrivere lo stato desiderato invece che i passi necessari per raggiungerlo. In K8S la descrizione dello stato viene aggiornata *applicando* dei file di tipo yaml chiamati manifest.

Di seguito prima progetteremo i manifest per ogni microservizio della nostra applicazione, tenendo conto dei singoli requisiti di funzionamento, le risorse e le dipendenze verso altri servizi presenti nel sistema; dopodiché studieremo un metodo per integrare questi manifest nel processo di CI/CD.

La prima cosa da fare progettando i manifest è quello di capire come mappare gli oggetti forniti da Kubernetes rispetto alla nostra idea di infrastruttura. Prima di procedere con un esempio di microservizio, esponiamo una rapida panoramica di cosa è possibile fare con gli oggetti base offerti da k8s [14]:

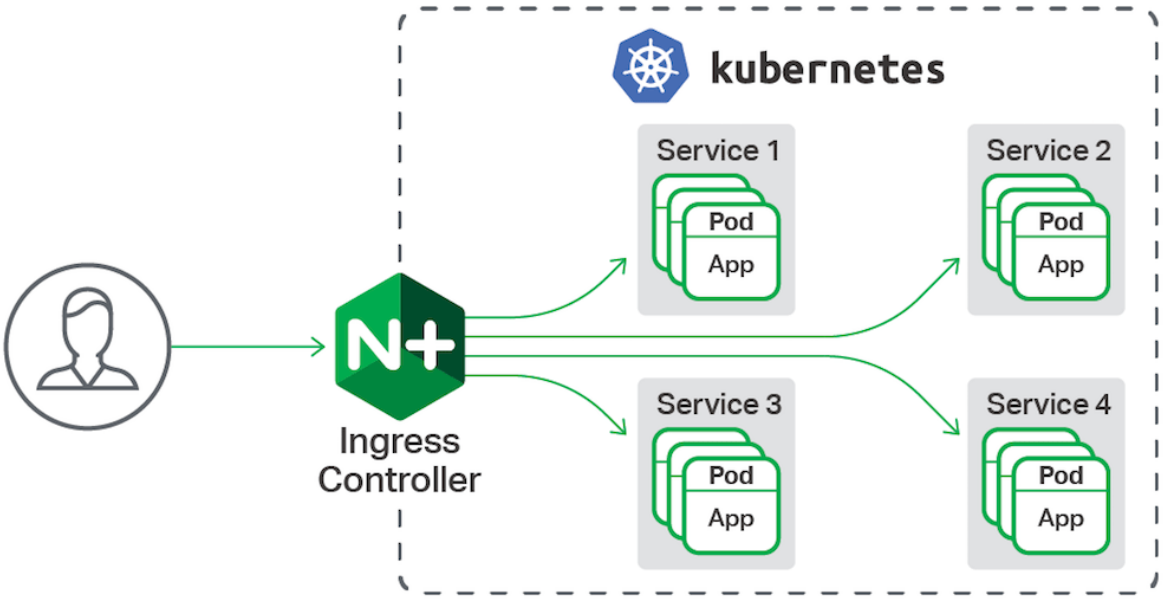

Figura 4.8: Organizzazione risorse di Kubernetes, fonte: nginx.com/blog

1. **Pod**: è la più piccola risorsa di computazione istanziabile in k8s. Rappresenta un gruppo di uno o più container di applicazioni e alcune risorse condivise (volumi). Funziona su una rete privata e isolata e di conseguenza i container possono dialogare tra loro utilizzando localhost. In genere è consigliabile mappare un solo container per pod, eccetto quando si vogliono adottare pattern più avanzati come quello del *side-car* [8].
2. **Deployment**: è un'astrazione di k8s responsabile della gestione (creazione, aggiornamento, eliminazione) dei pod. Per distribuire l'applicazione si possono utilizzare direttamente i pod descritti precedentemente, tuttavia, l'utilizzo di Deployments è consigliabile per le seguenti ragioni:
   (a) si ha un unico file manifest in cui vengono definite le specifiche dei pod in termini di quale immagine di container usare, repliche, risorse, persistenza e variabili di ambiente.
   (b) non devi preoccuparti di gestire i pod. Se uno dei pod termina, il controller assegnato al deployment ne creerà immediatamente un altro per rispettare il

numero di pod desiderato.
   (c) fornisce un meccanismo di *self-healing* in caso di guasto nel cluster. Se il nodo che ospita un Pod si interrompe o viene eliminato, il Deployment sostituisce l'istanza con un'istanza su un altro nodo del cluster.
   (d) fornisce un modo semplice per eseguire la strategia di aggiornamento dei pod. Di base viene utilizzata la tecnica *rolling* in cui l'aggiornamento avviene gradualmente e senza interruzioni di servizio, sostituendo un pod alla volta e spostando il traffico sulle repliche ancora attive.
3. **Service**: è un oggetto che fornisce un'astrazione per definire un set di Pod che forniscono logicamente lo stesso servizio e la relativa politica per accedervi. Di fatto implementa il meccanismo del Service Discovery (un'applicazione interessata ad un servizio accede tramite il suo nome piuttosto che utilizzando il singolo ip che identifica il suo pod) e Load balancer (su questo oggetto è possibile implementare politiche per pilotare su quale Pod indirizzare una richiesta, default round robin)
4. **Ingress**: è un semplice reverse proxy che instrada il traffico dal mondo esterno verso i servizi nel cluster. In ingress è possibile specificare, a fronte di un virtual host (hostname e eventuale path relativo), uno o più servizi a cui reindirizzerà il traffico.
5. **Secret**: consentono di archiviare e gestire informazioni sensibili, come password, token OAuth e chiavi ssh. Un deployment può assegnare un segreto al valore di una variabile di ambiente di un pod.
6. **Persistent Volume**: sono risorse di archiviazione che possono essere istanziate staticamente da un amministratore o dinamicamente su richiesta di un pod. Possiamo montare questi dischi sui Container in esecuzione dai pod per garantire che un'eventuale applicazione stateful possa conservare i propri dati rispetto alla ricreazione dei container che ne fanno parte. I Persistent Volume adempiono allo stesso scopo dei volumi in ambito Docker.

Ora che abbiamo definito i concetti con cui è possibile lavorare, esponiamo la strategia di implementazione. L'intento da raggiungere è una situazione simile a quella di Figura 4.8.

Innanzitutto per ogni microservizio costruiamo i seguenti manifest: *deployment.yaml*, *service.yaml* e *secrets.yaml*, che descrivono gli oggetti precedentemente esposti. Con tutti i microservizi si intendono anche i servizi forniti a livello aziendale, e già descritti nella Sezione 3.4, di cui fin'ora non abbiamo dovuto progettare il ciclo di rilascio (ocr, pdf2html e document categorization). Descrittori opzionali sono: *(i)* ingress.yaml solo per servizi che hanno bisogno di essere acceduti dall'esterno del cluster, vedi api gateway e client web; *(ii)* pvc.yaml per i servizi che necessitano di persistenza su volumi.

Analizziamo come caso pratico l'implementazione sviluppata per il microservizio dell'api gateway:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sisred-api
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: sisred-api
  template:
    metadata:
      labels:
        app.kubernetes.io/name: sisred-api
    spec:
      imagePullSecrets:
        - name: rd-registry
      containers:
        - name: sisred-api
          image: "psacr.maggiolicloud.it/sisred-api:735087598adbd"
          imagePullPolicy: Always
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: prod
            - name: SPRING_ELASTICSEARCH_REST_URIS
              value: "https://edb-es001.paas.elogic.cloud"
            - name: SPRING_ELASTICSEARCH_REST_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: elastic-credentials
                  key: password
            - name: APPLICATION_NLP_URL
              value: http://sisred-nlpserver
            - name: APPLICATION_PDF2HTML_URL
              value: http://sisred-pdf2html
          ports:
            - name: http
              containerPort: 8080
              protocol: TCP
          readinessProbe:
            httpGet:
              path: /management/health/readiness
              port: http
            initialDelaySeconds: 20
            periodSeconds: 15
            failureThreshold: 6
          livenessProbe:
            httpGet:
              path: /management/health/liveness
              port: http
            initialDelaySeconds: 120
          resources:
            limits:
              cpu: 2000m
              memory: 3Gi
            requests:
              cpu: 500m
              memory: 2250Mi
```

Listato 3: Kubernetes: api-deployment.yaml

Nel listato 3 che rappresenta l'oggetto deployments del microservizio (nel quale sono state rimosse alcune variabili di ambiente per mantenere leggibilità e privacy), possiamo notare dei problemi rispetto alla definizione del nostro flusso di rilascio:

1. riga 19: il tag dell'immagine docker è fissato mentre nella nostra pipeline dipende dallo SHA del commit che ha scatenato il rilascio.
2. le variabili di ambiente (riga 21), le risorse assegnate (riga 56) e le repliche (riga 6) sono statiche mentre sappiamo essere in funzione dei requisiti degli ambienti di destinazione.

Per risolvere questi problemi si potrebbe immaginare di avere una copia di tutti i manifest per ogni ambiente di rilascio, oppure di costruire un meccanismo di sostituzione dei valori utilizzando strumenti di base come *sed*. Nella realtà, essendo una delle problematiche più riscontrate nella gestione di queste soluzioni di rilascio, sono nati diversi strumenti che mirano a semplificare l'esperienza di utilizzo. Di seguito vedremo quello adottato dal team.

Helm

Helm[9] è un gestore di pacchetti per Kubernetes, immaginabile come l'equivalente K8S di yum o apt. Helm lavora distribuendo dei **Chart**, ovvero un pacchetto contenente una raccolta di tutte le risorse (manifesti YAML) dell'applicazione preconfigurate per essere eseguibili su Kubernetes tramite un semplice *install*.

All'interno di un Helm Chart, i precedenti manifest yaml sono decomposti in due parti: configurazione e valori. La parte di configurazione rimane dentro i file yaml e gestita tramite sintassi basata su Go template. La parte dei valori viene estratta e inserita in un file esterno ai template, chiamata *values.yml* e sarà utilizzata per generare i manifest reali prima di applicarli al cluster. Durante la fase di *helm install* i template vengono tradotti in file leggibili da kubernetes sostituendo le variabili Go Template con i valori presenti sia nel values.yml che in quelli forniti nella cli. Con questo meccanismo è possibile quindi scrivere un'unica configurazione che gestisce il rilascio di un applicazione e avere tanti file *values.yml* che pilotano i reali valori assegnati alle risorse descritte in quei template.

Riconducendolo alle necessità del sistema redazionale, si è quindi abbandonato l'utilizzo dei manifest classici in favore del meccanismo proposto da Helm. Per ogni microservizio si è creato un Helm Chart che avesse la configurazione di rilascio adeguata. Successivamente, basandoci sui tre ambienti a disposizione son stati generati i file dei valori con le relative specifiche delle risorse:

- values-review.yml
- values-test.yml
- values-prod.yml

Durante l'automazione, si tratterà di specificare il file values corretto per l'ambiente su cui si vorrà rilasciare il microservizio:

```
helm upgrade -i ${IMAGE_NAME} .helm/
      --create-namespace
```

[9] `https://helm.sh/`

```
--namespace ${K8S_NAMESPACE}
--set image.repository=${CI_REGISTRY}/${IMAGE_NAME}
--set image.tag=${CI_COMMIT_SHORT_SHA}
--set ingress.host.name=${APP_DOMAIN}
-f .helm/envs/values-${ENV}.yaml
```

4.8 Insfrastructure as Code

La progettazione dell'infrastruttura è la fase del ciclo di vita del software che definisce e configura tutte le componenti di sistema necessarie per il corretto funzionamento dell'applicazione. In Sezione 1.5.6 si sono già analizzati i motivi che da diversi anni suscitano largo interesse verso le pratiche di IaC.

Il sistema redazionale non fa eccezione e tra i fattori abilitanti che giustificano l'adozione di questa pratica [7] ve ne sono alcuni che mirano a risolvere specifiche problematiche emerse durante l'intervista ai membri del team riguardo i processi attuati nello sviluppo precedente (sezione 2.3):

- Creazione e configurazione dei sistemi in modo automatico, rapido e consistente, con conseguente riduzione dei tempi di gestione.
- Utilizzo della pratica di controllo di versione per far evolvere il codice che descrive l'infrastruttura insieme all'applicazione che supporta.
- Inserimento di uno strato di controllo qualità che possa validare in modo automatico anche l'infrastruttura che ospiterà l'applicativo.

Il team di sviluppo dapprima dovrà individuare uno strumento in grado di abilitare la pratica di IaC. Dopodiché userà tale strumento per tradurre la definizione dell'architettura in codice. Infine occorrerà progettare come attuare la creazione e configurazione di tutto il sistema durante il flusso di rilascio automatico, tenendo presente che ai diversi ambienti espressi in Sezione 4.2.3 sono associate diverse condizioni di lavoro. Le tipologie di ambiente da gestire saranno infatti di due tipi: *(1)* Ambienti stabili, ovvero sempre attivi e mai de-allocati (test e produzione); *(2)* Ambienti dinamici, ovvero creati quando necessario e di durata limitata nel tempo (review). Entrambe le tipologie condivideranno gli stessi componenti infrastrutturali, eventualmente parametrizzati in termini del quantitativo di risorse per cercare di bilanciare anche il fattore dei costi.

Prima di procedere con la vera e propria progettazione delle componenti di codice, è bene esporre le due proprietà fondamentali [20] che ci imponiamo di rispettare al fine di produrre un caso di successo:

- utilizzo della pratica dell'**infrastruttura immutabile**: per evitare scenari in cui la stessa configurazione iniziale diverge in tante piccole micro derivazioni (es: aggiornamento del sistema operativo in momenti temporali diversi), non sarà possibile modificare le componenti dell'infrastruttura. Sono ammesse solo le operazioni di creazione e distruzione. Aggiornare la configurazione di un nodo significa distruggerlo e ricrearlo con la configurazione aggiornata.
- nessun utilizzo promiscuo di automazione e interventi manuali: come vedremo successivamente, gli strumenti che permettono di effettuare IaC si basano sulla concezione di stato, che consiste in una rappresentazione di quello che il sistema pen-

sa essere la realtà. Agire manualmente invalida tale rappresentazione e provoca fallimenti gravi nel processo di manipolazione dell'infrastruttura.
- il codice che formalizza le componenti di infrastruttura non deve avere altro compito se non quello di istanziare ed eventualmente configurare le varie parti architetturali. Non è sua competenza, quindi, l'eventuale rilascio delle parti software che dipendono da essa (es: i container dei microservizi).

4.8.1 Scelta dello strumento

Il processo di scelta di uno strumento che permetta di adottare la pratica di IaC non è banale. Molti strumenti raggiungono le stesse funzionalità in modi tra loro molto diversi. I nomi che spiccano per importanza nello scenario Open Source sono ad esempio Terraform, Chef, Puppet, Ansible, SaltStack, CloudFormation e Openstack Heat. È necessario quindi effettuare un'analisi preliminare per capire quale strumento fa al caso nostro [7]:

- **gestione della configurazione o dell'infrastruttura**: sebbene entrambe le tipologie riescano a portare a termine gli obiettivi di entrambi i mondi, vogliamo adottare quello che più si adatta al nostro scenario. In particolare la maggior parte di configurazione del sistema avviene durante la progettazione dell'immagine docker dei nostri microservizi (es: scelta runtime java, impostazioni delle variabili di ambiente, eccetera). Se un giorno si deve aggiornare la versione di java per effettuare una patch di sicurezza, si effettuerà la build della nuova immagine docker con conseguente rilascio sul cluster. Per questo motivo risulta più naturale focalizzarsi su strumenti di gestione dell'infrastruttura.
- **gestione infrastruttura mutabile o immutabile**: abbiamo già parlato del motivo che ci spinge ad adottare il paradigma dell'infrastruttura immutabile.
- **linguaggio procedurale o dichiarativo**: nei linguaggi procedurali è compito del programmatore descrivere i passi necessari per arrivare alla configurazione desiderata. In quelli dichiarativi il codice esprime sempre lo configurazione desiderata del sistema. Sarà poi compito dell'interprete capire come raggiungere tale stato. Alcuni vantaggi di adottare uno stile dichiarativo sono: a) il programmatore si concentra sul cosa invece del come; b) il codice rispecchia lo stato attuale del sistema; b) si incentiva il riuso del codice in quanto il codice scritto non deve tenere in considerazione lo stato del mondo reale e può quindi essere scritto in modo più generico.
- **comunità**: la IaC è una pratica DevOps relativamente giovane. Per questo motivo nel contesto del sisred si darà maggior rilevanza alla base di utenza attiva rispetto all'età dello strumento. Questo ci permetterà di avere maggior supporto e trovare documentazione più rilevante.

	Linguaggio	**Tipo**	**Infrastruttura**	**Comunità**
Chef	Procedurale	Config Mgmt	Mutabile	Grande
Puppet	Dichiarativo	Config Mgmt	Mutabile	Grande
Ansible	Procedurale	Config Mgmt	Mutabile	Grande
SaltStack	Dichiarativo	Config Mgmt	Mutabile	Media
OpenStack Heat	Dichiarativo	Provisioning	Immutabile	Piccola
Terraform	Dichiarativo	Provisioning	Immutabile	Media

Tabella 4.2: Sommario dei principali strumenti di IaC [7].

Il team del sistema redazionale ha ricercato uno strumento open source, agnostico dal cloud, che includesse il paradigma dell'infrastruttura immutabile e che fosse supportato da una comunità attiva. Come si vede bene dal sommario in Tabella 4.2, lo strumento che aderisce meglio a questa definizione è senza dubbio **Terraform** (conosciuto anche come TF).

Funzionamento

Terraform è un strumento open source scritto in Go Lang e rilasciato dall'azienda Hashicorp[10]. Tramite la *terraform-cli* è possibile effettuare il rilascio dell'infrastruttura desiderata a partire da qualsiasi macchina (dal portatile dello sviluppatore o dal server che effettua la CI/CD). Dietro le quinte, a partire da dei file di configurazione chiamati Configuration File (.tf), Terraform saprà quale API (e in quale sequenza) dei vari cloud provider invocare per effettuare il processo di creazione. Questi file di configurazione sono scritti in un linguaggio proprietario di Hashicorp chiamato *hcl* e, di fatto, consentono di descrivere l'architettura della propria applicazione astraendo dalle singole interfacce pubbliche rilasciate dai cloud provider.

Per abilitare l'interfacciamento da terraform, i vari fornitori rilasciano un pacchetto che prende il nome di **provider**. Un provider non è altro che una serie di definizioni e documentazione delle varie risorse istanziabili tramite le relative API. Una volta importato, terraform utilizzerà il provider sia per validare i file di configurazione, sia per sapere come tradurre le definizioni in chiamate reali.

Una buona pratica in uso da Terraform è quella di presentare l'anteprima delle modifiche allo sviluppatore, in modo che possa validare e intercettare eventuali modifiche prima che raggiungano produzione. Questa pratica è implementata sottoforma di *plan* file, un vero e proprio file di testo che contiene il delta delle modifiche che terraform vuole mettere in pratica.

Un ultimo concetto introdotto da Terraform è quello dello **stato**. Ogni volta che TF viene eseguito, aggiorna un file chiamato *Terraform State file* (.tfstate) che non è altro che un Json che tiene traccia della corrispondenza tra le risorse codificate nei configuration file e le risorse istanziate nel mondo reale. Questo stato può essere gestito sia localmente che in modalità remota. Nel sistema redazionale sarà utilizzato in modalità remota, abilitando così la gestione dell'infrastruttura su più macchine di sviluppo e all'interno della pipeline di CI/CD.

Successivamente, quando qualcuno del team dovrà effettuare una modifica all'infrastruttura, invece che eseguire manualmente le modifiche, gli basterà aggiornare i Terraform configuration file, validarli tramite test automatici (terraform validate) e code review (terraform plan), e infine applicare le modifiche tramite *terraform apply*. Lo strumento analizzerà il proprio file di stato per organizzarsi sul **come** procedere per raggiungere la configurazione desiderata.

[10] `https://www.terraform.io/`

4.8.2 Codifica dell'infrastruttura

Per la codifica dell'infrastruttura in Terraform Configuration Files, si parte dai singoli componenti emersi durante l'analisi dei microservizi alla sezione Sezione 3.1. A queste definizioni base si aggiungeranno altre più specifiche che dipendono: *(i)* dal tipo di configurazione che vogliamo raggiungere; *(ii)* dal cloud provider del singolo componente; *(iii)* dall'ambiente che stiamo generando (es: test o produzione).

Per fare un esempio di quanto appena scritto, pensiamo agli ambienti review e master. Nel primo caso voglio creare un ambiente da zero con un dominio raggiungibile pubblicamente e sotto certificato HTTPS. Nel secondo invece devo integrare un dominio e certificato HTTPS prestabiliti e acquistati precedentemente. Questa casistica si risolve affiancando alla configurazione di AKS un nuovo file che descrive la necessità di avere, sul cluster appena istanziato, il plugin del famoso servizio *Lets Encrypt*[11] per la generazione automatica dei certificati. Questo nuovo file sarà preso in considerazione solo con opportune configurazione di lancio di Terraform (che discriminano l'ambiente destinazione).

Di seguito i componenti infrastrutturali derivati dalla scomposizione in microservizi:

- **Azure Resource Group**: Su Azure ogni componente deve risiedere all'interno di un gruppo di risorse che ne determina il raggruppamento logico e l'area geografica in cui istanziarlo (es: nord europa)
- **AKS**: è l'implementazione del cluster kubernetes fornita da Azure. Qui sopra verranno rilasciati i container dei vari microservizi. Occorrerà quindi:
  - configurare il numero ed il tipo di nodi del cluster
  - configurare la scalabilità automatica
  - utilizzare l'implementazione nginx di Ingress[12]
  - preparare i namespace necessari per ospitare lo stack applicativo (i microservizi che verranno poi rilasciati)
  - configurare gli opportuni segreti
- **Azure Storage**: è il servizio di storage che utilizzerà l'api gateway.
- **Azure Service Bus**: è il servizio di messaggistica asincrona utilizzato da tutti i microservizi.
- **Azure SQL Server**: è il database relazionale in uso dall'api gateway.
- **Elastic on Azure**: è il servizio di ricerca full text.
- **Azure Monitor Service**: è il servizio di monitor offerto da Azure. Vedremo la relativa analisi nei prossimi capitoli.

Gestione dei segreti

Tutti i componenti infrastrutturali sono istanziati con credenziali di accesso casuali, in modo da evitare di inserire nel controllo di versione informazioni sensibili. Questi valori (stringhe, file di configurazione, token, eccetera) saranno necessari per il corretto accesso da parte di servizi terzi. Sarà fondamentale salvare questa configurazione per informare opportunamente lo stack applicativo che verrà rilasciato in un secondo momento.

La modalità più naturale con la quale trasmettere queste informazioni per essere usate in un secondo momento (ad esempio dai microservizi dell'applicativo) è quella di persisterle

[11] `https://letsencrypt.org/`
[12] `https://kubernetes.github.io/ingress-nginx/`

tramite gli oggetti *secret* di kubernetes. I manifest di kubernetes dei microservizi saranno quindi configurati opportunamente per associare i valori dei segreti alle variabili d'ambiente del container. Di conseguenza i microservizi rimangono agnostici dalla modalità di creazione dell'infrastruttura, a patto che si converga sempre sull'utilizzo dei segreti.

Gestione degli ambienti

Essendo in un contesto cloud, ogni servizio ha la possibilità di definire dei livelli di servizio, che spesso si materializzano in affidabilità, prestazioni e risorse assegnate. Questi livelli son soggetti a costi che possono risultare incisivi sulla gestione del progetto. Ad esempio nel momento della scrittura i nodi del cluster di kubernetes di tipo *Standard B2s* costano circa 30 €/mese contro gli 85 €/mese della tipologia *Standard D2*. Su un cluster di 4 nodi accesi per un anno abbiamo una differenza di circa 2500€.

È evidente che gli SLA[13] richiesti da un'infrastruttura di produzione non siano gli stessi di quelli richiesti per l'ambiente di review. Rimane tuttavia valida la regola per cui si vuole mantenere tutte le configurazioni il più identiche possibili, in modo da intercettare le stesse tipologie di anomalie.

Per raggiungere tale obiettivo, introduciamo la parametrizzazione dei Terraform Configuration File, con delle variabili specificate in file che rappresentano il tipo di ambiente di destinazione. In terraform questi file sono chiamati *.tfvars* e vanno specificati durante la fase di plan.

```
terraform plan -out=my.tfplan -var-file=tfvars/dev.tfvars
```

Fatto ciò, l'ultima gestione da mettere in pratica sarà quella dello stato. Dovendo gestire tanti ambienti avremo quindi la necessità di dover gestire altrettanti stati. Questa gestione è fondamentale per evitare di fare pianificazioni errate ed evitare il rischio di distruggere in modo irreversibile pezzi di infrastruttura dell'ambiente sbagliato.

In questo frangente, per evitare una gestione manuale, terraform (a partire dalla versione 0.10) ha introdotto il concetto di **workspace**. Ogni workspace isola uno stato completo di Terraform e grazie a questa direttiva è possibile quindi associare stati multipli alla stessa configurazione di IaC. Per il modello di branching del sistema redazionale, in cui gli ambienti sono rappresentati da rami git, risulta naturale utilizzare la convenzione che lo stato di un ambiente risiede in un workspace che ha lo stesso nome del ramo associato a tale ambiente.

Riusabilità del codice

Codificando l'infrastruttura capita di frequente di dover descrivere gli stessi componenti, con solo alcuni parametri diversi. Si pensi ad uno scenario a microservizi dove ognuno lavora con un proprio database relazionale istanziato nel cloud e con diverse configurazioni di firewall per pilotarne l'accesso. Così facendo potremo facilmente trovarci nella situazione in cui si hanno numerose configurazioni .tf il cui numero di righe di codice duplicate rendono difficile la comprensione ed il refactory del codice.

[13]Service-level agreement

Nei linguaggi di programmazione classici (come Java, Python, Ruby, ecc), quando siamo in una situazione del genere, il codice viene estratto in una funzione con una firma ben definita e riutilizzata in varie parti del codice.

In terraform si ottiene lo stesso risultato utilizzando il concetto di **modulo**. Un modulo non è altro che un pezzo di infrastruttura richiamabile con un'interfaccia ben definita e che produce in *output* una serie di informazioni. Nell'esempio precedente restituirà la stringa di connessione per poter accedere ai vari database istanziati.

Oltre a gestire moduli creati localmente, è possibile referenziarne altri in modo remoto (es: presenti su un registro). In questo modo, un'organizzazione che vuole rendere omogenee le configurazione dei vari pezzi architetturali, potrà fornire questi moduli ai progettisti di IaC che li utilizzeranno come *blackbox* per comporre la propria infrastruttura. Nel caso di studio del sistema redazionale, son stati generati due moduli: uno per il cluster di kubernetes (eventualmente per la distribuzione di un cluster standard a livello aziendale) e uno per la gestione dei database (riuso del codice nel caso di un database per microservizio).

4.8.3 Automazione

Ora che abbiamo progettato tutti i tasselli della pratica di IaC, vedremo come integrarli nel ciclo di rilascio del sistema redazionale. Prima di tutto occorre capire in quali fasi della pipeline è bene inserire i passi di IaC. Ricordiamo che per Terraform saranno in sequenza: validate, plan e apply. Tra di loro devono essere sequenziali ma rispetto agli altri processi della pipeline possono essere eseguiti parallelamente. L'unico punto di raccordo sarà necessario nel momento in cui la pipeline dovrà eseguire la creazione dei microservizi. Per quel momento il rilascio dell'infrastruttura dovrà essere già avvenuto poiché sappiamo che i microservizi saranno ospitati dal cluster kubernetes, la cui creazione è in carico alla fase di IaC.

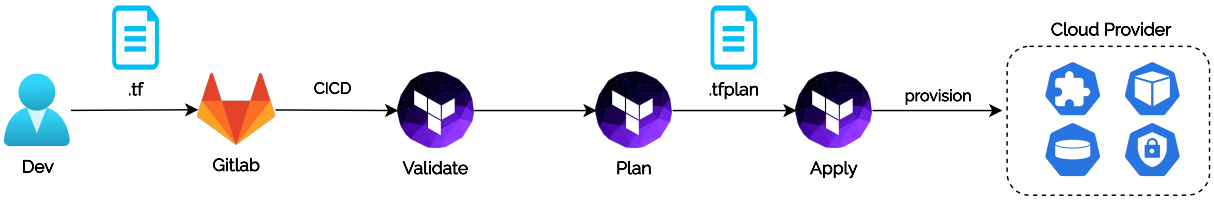

Figura 4.9: Automazione Terraform in CI/CD

Il flusso consiste nei passi mostrati in Figura 4.9:

1. Il programmatore aggiorna i file .tf che rappresentano l'infrastruttura e li pubblica sul sistema di controllo di versione (git).
2. Git scatena la CI/CD per il ramo git aggiornato.
3. **tf-validate** si assicura che i file .tf siano scritti correttamente.
4. **tf-plan**, a partire dallo stato associato al workspace (dove nome workspace equivale al ramo git in esecuzione), genera un piano di modifiche .tfplan.
5. **tf-apply**, applica le modifiche specificate nel .tfplan precedente e aggiorna lo stato nel terraform workspace. Per consentire di effettuare code review sul .tfplan generato è possibile richiedere l'avvio manuale di questo passo.

6. La pipeline può proseguire e rilasciare anche lo stack applicativo sopra l'infrastruttura appena istanziata.

La parte più complessa consiste nel progettare la pipeline in modo che recepisca dinamicamente le configurazioni per accedere al cluster, in modo da poter rilasciare i container direttamente sul cluster appena creato senza bisogno di alcun intervento manuale. Durante la progettazione dei terraform configuration file ci basta produrre in output il file *kubeconfig* che contiene tutti i dati sensibili per connettersi al cluster. Vedremo successivamente nella fase di progettazione della pipeline come sfruttare questo output per istruire correttamente *helm install* della fase di deploy.

4.8.4 Test infrastrutturali

Produrre test per la validazione dell'infrastruttura non è un compito banale. La principale complicazione deriva dal fatto che, dietro le quinte, i Terraform configuration file non sono altro che un modo alternativo per effettuare chiamate alla API dei singoli cloud provider [7]. In questo scenario, qualsiasi tipo di mock[14] invalida il test stesso, poiché quello che si vuole testare spesso è l'interazione col provider stesso. Per questo motivo in genere conviene ripiegare sulla strada più semplice: istanziare l'architettura e valutare se qualcosa è andato storto.

Nell'ottica del sistema redazionale questa filosofia si traduce in: gli ambienti di review e test intercettano eventuali problemi nella codifica dei terraform configuration file. Se la creazione avviene con successo, si avrà un feedback sull'effettivo funzionamento sia da un livello di monitoraggio (che vedremo in Sezione 4.9), sia dai test e2e (Sezione 4.5.2) che mirano a stimolare e validare ogni strato applicativo e quindi a intercettare anche problemi sul piano infrastrutturale.

4.9 Monitoraggio

Durante lo studio dello stato dell'arte, Sezione 1.5.7, si sono già analizzati i vantaggi di adottare un approccio continuo di analisi dello stato attuale del sistema. Questa analisi può avvenire su più livelli: ad esempio possiamo analizzare come si sta comportando la pipeline di CI/CD, come performa l'ambiente di sviluppo dei singoli programmatori o ancora meglio come si sta comportando il sistema in produzione.

Mentre per i primi due scenari in esempio spesso si ha supporto diretto dalla piattaforme in uso (es: segnalazione pipeline non funzionante, dashboard con report della durata del ciclo di CI/CD), in questo capitolo vogliamo prendere in considerazione l'analisi del comportamento del sistema.

Questo tipo di analisi permette di essere proattivi nell'identificare e risolvere sia problemi infrastrutturali sia anomalie di tipo software prima che influenzino l'ambiente di produzione finale, e quindi, il cliente [10]. L'identificazione di una problematica avviene analizzando, manualmente o automaticamente, dei flussi di dati che contengono valori riguardo parametri significativi del sistema. Questi dati prendono il nome di **dati di telemetria**.

[14] Tipologia di test in cui parte della logica è simulata con servizi *fittizi*

In un contesto come quello del sistema redazionale, dove la generazione degli ambienti è automatizzata e pressoché nessun passo della pipeline è manuale, diviene fondamentale avere uno strumento in grado di capire come si comporta il sistema e fornire un canale di retroazione per migliorare la qualità dell'applicativo.

4.9.1 Scelta dello strumento

Nel progettare il sistema di monitoraggio consideriamo le seguenti caratteristiche:

- praticità di integrazione: si intente quanto è invasivo l'intervento all'applicazione per renderla conforme all'esportazione dei dati di telemetria verso lo strumento adottato.
- dashboard e allarmi: il sistema deve permettere sia la creazione di dashboard personalizzate, sia deve dare supporto alla gestione di avvisi verso specifici canali di comunicazione.
- conservazione dei dati: se ci sono limiti nella conservazione dei dati. Nell'era dei big data, avere uno storico completo dei dati di telemetria, abilita algoritmi di Machine Learning come l'analisi predittiva.
- costo: sia in termine finanziario che in quello di consumo delle risorse. Anche per piccole/medie applicazioni la mole dei dati di telemetria può raggiungere volumi considerevoli. Lo strumento deve essere in grado di collezionare e analizzare grandi quantità di dati.

La ricerca della soluzione è avvenuta valutando due strumenti:

1. **Prometheus**[15]: è uno dei progetti open source più conosciuti. Di fatto è scomposto in tanti moduli e tra questi vi è il sistema di Alert chiamato *AlertManager*. Per la visualizzazione dei dati in dashboard occorre adottare uno strumento di visualizzazione separato come *Grafana*. Nel cloud di riferimento del sisred non è previsto una versione *gestita* del servizio di prometheus; di conseguenza il team decide di installarlo come plugin helm all'interno del cluster kubernetes.
2. **Azure Monitor**: è un servizio di monitoraggio gestito dal cloud provider. Per la visualizzazione della dashboard senza l'utilizzo di componenti esterni (tipo Grafana del punto precedente) è necessario navigare sulla console Cloud. Il servizio integra il supporto ad alert verso canali multipli e la conservazione dei dati dipende dal livello di contratto (e quindi di costo) attivato.

Praticità di integrazione

Seppur entrambi gli strumenti abbiano un livello di integrazione poco invasivo, in linea con le pratiche dell'integrazione *zero code*, lo raggiungono con modalità diverse. In entrambe le soluzioni infatti non ci sarà bisogno di effettuare modifiche applicative ai microservizi, che non sapranno di essere monitorati.

Prometheus, nell'implementazione a container installata sul cluster k8s, implementa il pattern **sidecar**. Un sidecar è un componente software istanziato sulla stessa rete dell'applicazione principale ma eseguito sul proprio processo o contenitore. Generalmente le funzionalità di business vengono implementate nel container principale mentre quelle

[15] `https://prometheus.io/`

di utilità nel container sidecar, che spesso si interpone come proxy anche per il traffico di rete tra il container principale e gli altri servizi [17]. Di conseguenza il sidercar riesce ad arricchire con funzionalità aggiuntive il container che affianca, rimanendo tuttavia agnostico dalla specifica tecnologia con la quale è implementato.

Di fatto tramite questo pattern, nel contesto del monitoraggio, il sidecar riesce ad esportare le metriche del container dell'applicazione principale verso il proprio server che colleziona i dati di telemetria.

Nel corrispettivo servizio di Azure, poiché il cloud provider gestisce direttamente il cluster k8s, l'integrazione è ancora meno invasiva; una volta abilitato il monitoraggio, Azure si preoccuperà di istanziare automaticamente i componenti infrastrutturali (dentro e fuori dal cluster) in grado di esportare le metriche verso un servizio che colleziona e immagazzina i dati. In questo caso il processo di esportazione delle metriche è molto trasparente rispetto sia alla gestione dall'applicazione che a quello dell'infrastruttura.

Dashboard, interrogazioni e allarmi

Entrambi gli strumenti offrono un proprio linguaggio di interrogazione per il recupero e visualizzazione dei dati.

Nella sperimentazione fatta su Prometheus è necessario includere due ulteriori moduli: 1) Grafana per la gestione di dashboard personalizzate; 2) AlertManager per la gestione degli allarmi. In Azure invece è possibile effettuare la totalità delle operazioni dalla console dello sviluppatore.

I risultati ottenuti sono circa gli stessi. Da un lato con Prometheus e Grafana, avendo una comunità molto attiva, è possibile trovare sul marketplace numerose dashboard e interrogazioni per il monitor di infrastrutture a container. Dall'altro lato in Azure la procedura guidata di creazione di dashboard rende immediata la generazione di grafici e tabelle, evitando di dover ricorrere a soluzioni sviluppate da terzi.

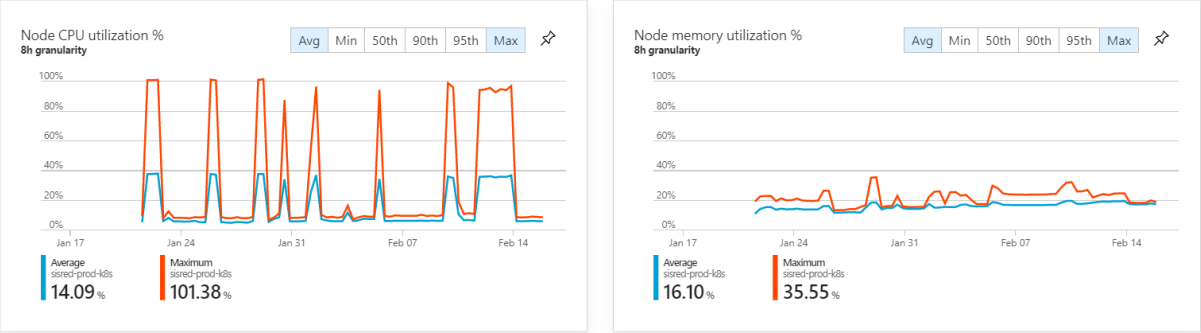

Figura 4.10: Esempio di grafici estratti da Azure Monitor

Conservazione dei dati e costi

Per entrambi i sistemi non ci sono limiti sulla mole di dati conservati. Indirettamente questo volume inciderà sul costo del servizio. Per Prometheus infatti la gestione delle risorse avviene direttamente dal cluster k8s su cui è installato. La necessità di dover scalare Prometheus significa consumare più risorse hardware e quindi investire in un

cluster più performante. In Azure Monitor, invece, l'intero servizio è *gestito* e sia la scalabilità che la gestione delle risorse non è in carico al team del sistema redazionale. Tuttavia il costo del servizio dipende sia dal volume di dati trasmesso sia dalla durata di conservazione che si vuole adottare.

In questo caso è risultata più decisiva la gestione cloud rispetto a quella di Prometheus. Infatti da test effettuati dal team del sisred, in un installazione del cluster di k8s effettuata su un ambiente di produzione standard, il 40% delle risorse erano allocate per mantenere attivo il server di Prometheus. Inoltre, nel servizio cloud offerto da Azure viene promosso il modello di utilizzo *Pay As you Go*, ossia un modello di business che fattura solo sull'effettivo utilizzo del servizio, riducendo di molto i costi dedicati al monitoraggio.

Decisione

I due strumenti permettono di raggiungere risultati pressoché equiparabili. Tuttavia per il sistema redazionale la facilità d'integrazione, di costruzione delle dashboard e il notevole risparmio sui costi di un'infrastruttura dedicata hanno condizionato l'abbandono di Prometheus in favore del servizio gestito dal Cloud Provider: Azure Monitor.

4.9.2 Automazione

Nell'automatizzare il servizio di monitoraggio occorre prendere in considerazione due tematiche: 1) come gestire la propagazione di un allarme dal sistema di monitoraggio al team di sviluppo; 2) come integrare l'infrastruttura di monitoraggio nel flusso di CI/CD che stiamo ideando.

Gestione di un allarme

In Azure Monitor è possibile definire uno o più *action group*. Questi sono gruppi di azioni da eseguire in corrispondenza all'attivazione di un allarme. Un gruppo di azioni non è altro che una collezione di preferenze di notifica in cui vengono definiti i canali di comunicazione degli allarmi. Basandoci sulla gravità dell'allarme, è possibile definire diversi gruppi di azioni.

Nel caso del sistema redazionale si identificano due livelli di gravità:

- avviso: il problema riscontrato non compromette la stabilità sistema ma va preso in esame il prima possibile.
- critico: il problema compromette la stabilità del sistema e va risolto subito.

A livello avviso è stato associato l'invio di un messaggio sul workspace slack dove il team comunica e collabora tutti i giorni. In particolare è stato predisposto un canale dedicato alla ricezione degli avvisi dal sistema di monitoraggio. Per la configurazione è bastato creare un *webhook* da slack e riportarlo nel gruppo di azioni di azure.

A livello critico invece, oltre alla notifica sul canale slack, si è predisposto l'invio di un email a tutti i membri del team, in modo da avere una reazione più celere al problema.

```
resource "azurerm_monitor_action_group" "critical" {
  name                = "sisred-main-actiongroup"
  resource_group_name = var.resource_group_name
  short_name          = "sisredalerts"
  email_receiver {
    email_address = "alessandro.neri12@studio.unibo.it"
    name          = "sendtoDevOps"
  }
  webhook_receiver {
    name        = "slackChannel"
    service_uri = "http://example.com/alert"
  }
}
resource "azurerm_monitor_metric_alert" "restarting_container_alert" {
  name                = "Restarting container count"
  resource_group_name = var.resource_group_name
  scopes              = [azurerm_kubernetes_cluster.this.id]
  description         = "Triggered when Restarting container count > 0"
  criteria {
    metric_namespace = "Insights.Container/pods"
    metric_name      = "restartingContainerCount"
    aggregation      = "Average"
    operator         = "GreaterThan"
    threshold        = 0
  }
  frequency   = "PT30M"
  window_size = "PT30M"
  action {
    action_group_id = azurerm_monitor_action_group.critical.id
  }
}
```

Listato 4: Terraform - Definizione del gruppo di azioni e allarme critico

Rilascio

Tutti i componenti del sistema di monitoraggio sono da considerare come infrastruttura di supporto al software applicativo. In quanto tale si decide di integrare tutta la logica di creazione e integrazione nei flussi di IaC visti in Sezione 4.8.

In particolare le risorse che si dovranno codificare nei Terraform Configuration file sono: 1) il servizio Azure Monitor; 2) gli allarmi associati a particolari interrogazioni sulle metriche; 3) le dashboard di consultazione.

Nel listato 4 si vede un estratto della soluzione implementata. L'esempio codifica, tramite Terraform Configuration file, la creazione di un allarme critico quando nella finestra temporale degli ultimi 30 minuti si è riavviato almeno un container. In tale scenario si noti il gruppo di azioni che notifica i programmatori sia via email sia attraverso il canale di progetto su slack.

Capitolo 5

Implementazione della pipeline

Nelle sezioni precedenti abbiamo visto i passi che comporranno il ciclo di rilascio automatico del sistema redazionale. Di seguito, metteremo in pratica i concetti della Sezione 1.5.8, in particolare scegliendo quale sistema di CICD adottare e traducendo le specifiche di automazione in un file di pipeline che possa essere consumato da tale sistema.

5.1 Piattaforma

Nella ricerca di una piattaforma software di CI/CD uno dei fattori fondamentali per determinare la scelta consiste nel valutare se adottare una soluzione gestita (anche detta *in-house*, *on-premises*) oppure in servizio cloud. Questa scelta non è solo determinata dalle modalità di gestione operative (aggiornamento, backup, scalabilità, eccetera), ma anche da tematiche che spaziano tra: costi, politiche sulla riservatezza del codice, sulla sicurezza, affidabilità del servizio, flessibilità d'integrazione con altri strumenti aziendali e limiti imposti dal tipo di licenza.

	Gestione in cloud	**Gestione in-house**
Strumenti	Gitlab, Azure, Bitbucket	Gitlab, Jenkins
Costo di avvio	Basso	Alto
Sicurezza dei dati	Controllo limitato	Alto controllo
Costo di gestione dei dati	Basso	Alto
Flessibilità di personalizzazione	Molto rigida	Alta
Affidabilità	Basata sul provider	Basata sul personale

Tabella 5.1: Tabella comparativa: CI/CD su cloud e in-house [21].

Tuttavia le modalità con cui i servizi cloud necessitano di accedere ai sorgenti, in modo da fornire una piattaforma gestita di CI/CD, invalidano i vincoli tecnologici imposti a livello aziendale (Sezione 2.5.2). Generalmente l'accesso avviene in una di queste modalità:

- prendendo in gestione il repository git (in modo diretto o indirettamente tramite una copia sincronizzata): vìola il requisito sulla riservatezza dei sorgenti.

- utilizzando un repository git fornito dall'azienda, a patto che questo sia visibile dai loro server: vìola il requisito sulla sicurezza in quanto il repository git deve rimanere privato.

Queste considerazioni hanno portato il team del sisred a optare per un sistema di CI/CD on premise; seppur il panorama IT sia frammentato da una vasta gamma di strumenti, ognuno dei quali con le proprie peculiarità e sopratutto con la propria modalità e sintassi per descrivere il ciclo di automazione, si sceglie di adottare le **Gitlab Pipeline**. Il motivo principale risiede nel fatto che Gitlab è già la piattaforma di controllo di versione di riferimento per l'intera azienda; utilizzare quindi il suo sistema di CI/CD ne comporta una migliore integrazione con tutte le funzionalità già usate per il flusso di sviluppo (es: issue, merge request, report).

5.2 Concetti

In Sezione 1.5.8 si è già introdotto in cosa consiste l'utilizzo di una pipeline per automatizzare il ciclo di rilascio. Vedremo di seguito come ricondurre quelle nozioni sui concetti forniti da Gitlab.

GitLab CI/CD è configurato da un file chiamato *.gitlab-ci.yml* posto nella radice del repository di progetto. Le pipeline sono costituite da una o più fasi (direttiva **stages**) che vengono eseguite in ordine e ciascuna può contenere uno o più **processi** eseguiti in parallelo. I processi (direttiva **Job**) contengono le specifiche di *cosa* deve essere fatto e possono essere eseguiti solo dall'agente GitLab Runner[1].

Al verificarsi di un **evento scatenante** (commit, merge request, tag, eccetera) Gitlab analizza il repository alla ricerca del descrittore della pipeline. Successivamente estrae i processi da eseguire e li distribuisce ai Gitlab Runner nell'ordine e parallelismo specificato dagli stage.

Gitlab Runner oltre a eseguire i comandi specificati dal Job, è incaricato di preparare l'ambiente isolato in cui verrà immerso il processo. Raggiunge tale scopo tramite direttiva **image** e **variables**: la prima serve a specificare il nome dell'immagine docker in cui istanziare il processo; la seconda fornisce delle variabili d'ambiente utili per parametrizzare il comportamento del container che esegue il processo (segreti, tag, informazioni dinamiche come il commit hash).

In Gitlab i vari ambienti isolati creati per gestire i processi non condividono alcuna informazione. Per questo motivo quando si vuole propagare un artefatto intermedio per la successiva raffinazione da parte di altri job, si utilizza la direttiva **artifacts**. Questa, oltre a propagare file o cartelle tra i vari container dei job, istruisce Gitlab per considerare quei file *rilevanti* nel contesto del progetto e quindi consultabili nella schermata riepilogativa delle pipeline. Oltre a file binari intermedi un artefatto può essere segnalato come file di reportistica: junit, copertura dei test, qualità del codice. Gitlab utilizza questi report sia per valorizzare opportune dashboard sia per implementare controlli di qualità sulle attività di merge-request.

[1] Un componente di Gitlab designato all'esecuzione di processi durante la CI/CD

5.3 Struttura della pipeline

Ora che abbiamo definito i concetti base della pipeline, non ci resta che raccogliere il risultato di tutta la progettazione dell'automazione svolta precedentemente. Le naturali relazioni di dipendenza si codificano in stages differenti della pipeline, mentre processi che svolgono la stessa funzione logica vengono parallelizzati all'interno dello stesso stage. L'ordine degli stage è definito da due fattori: *(i)* i processi di uno stage si basano su artefatti generati in stage precedenti; *(ii)* i processi di uno stage devono essere esegui solo se quelli precedenti son terminati con successo (es: controllo di qualità). Di seguito un albero gerarchico dei dati raccolti, dove i nodi radice indicano gli stage identificati:

1. **build**
   (a) api-build
   (b) web-build
   (c) nlp-build
2. **test**
   (a) api-test
   (b) web-test
   (c) nlp-test
3. **analysis**
   (a) api-sonar
   (b) web-sonar
   (c) nlp-sonar
   (d) tf-validate
4. **package**
   (a) api-package
   (b) web-package
   (c) nlp-package
5. **release**
   (a) api-push
   (b) web-push
   (c) nlp-push
6. **cve-scan**
   (a) api-trivy
   (b) web-trivy
   (c) nlp-trivy
7. **iac-plan**
   (a) tf-plan
8. **iac-apply**
   (a) tf-apply
9. **iac-config**
   (a) iac-config
10. **deploy**
    (a) api-deploy
    (b) web-deploy
    (c) nlp-deploy
    (d) pdf2html-deploy
    (e) doccat-deploy
    (f) tika-deploy
11. **e2e**
    (a) cypress-e2e
12. **iac-destroy**
    (a) tf-destroy

Questa gerarchia rappresenta la totalità dei passi disponibili per la CI/CD. Vedremo successivamente gli scenari che determinano un avvio parziale dei processi della pipeline. Si noti inoltre, come previsto dall'analisi, che la CD di software integrato dal lavoro di terze parti (pdf2html,doccat,tika) compare solo nello stage di effettivo rilascio chiamato 10-deploy.

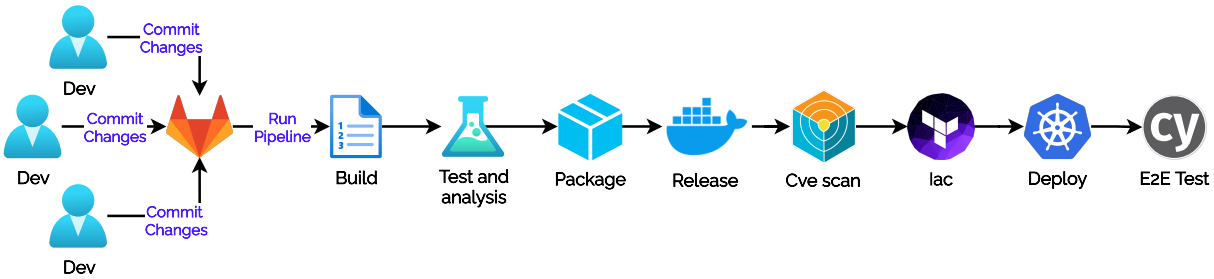

Figura 5.1: Rappresentazione grafica dell'intera pipeline di CI/CD

5.3.1 Condizioni di attivazione

In Sezione 5.3 si è definita la totalità di passi che compongono la pipeline del sisred. È tuttavia verosimile che non sia sempre necessario l'attivazione di tutti i passi. Per discriminare quando è opportuno attivare un job della pipeline occorre tenere in considerazione sia il tipo di evento che ha scatenato la pipeline, sia la semantica associata al ramo git

origine dell'evento, in modo da poter identificare se siamo in una condizione di CI oppure di CI/CD.

Per pilotare su quali eventi scatenare il singolo processo che compone una pipeline, Gitlab fornisce la direttiva **rules**. Nel contesto del sisred reagiamo a due situazioni: pubblicazione di un **commit** e richiesta di **merge request**.

CI

I passi associati alla pratica di CI sono quelli che mirano a validare la bontà delle modifiche introdotte dallo sviluppatore, con lo scopo d'identificare il prima possibile diversi livelli di problematiche: build, test, analysis e package.

Questi passi devono essere eseguiti in corrispondenza sia di un commit sia di una merge-request, a prescindere dal branch di origine. L'unica eccezione è fatta per i job descritti dalle voci da 3a a 3c che riguardano l'analisi statica con sonar e, come già detto precedentemente, verranno eseguite solo quando il ramo git di origine equivale a develop.

CI/CD

La pratica di CI/CD entra in esecuzione quando è necessario che la modifica al codice si propaghi anche sugli ambienti di produzione; perciò dopo aver eseguito i passi di CI, la pipeline prosegue fino ad includere il resto dei processi definiti in sezione 5.3 e che contengono la logica di pubblicazione e distribuzione delle immagini docker dei vari microservizi. Definiamo ora i vari comportamenti a seconda dell'ambiente che si dovrà aggiornare:

- **review**: scatenato in presenza di una merge-request e quando il ramo di origine ha nome nel formato *review-** e prevede l'esecuzione di tutti i passi previsti.
- **test**: scatenato quando il ramo di origine è *test* e prevede l'esecuzione di tutti i passi tranne il tf-destroy (12a).
- **produzione**: scatenato quando il ramo di origine è *master* e prevede l'esecuzione di tutti i passi tranne il tf-destroy (12a).

In tutti gli scenari le definizioni dei job descritte nello stage 10-deploy, per comunicare con il cluster k9s del relativo ambiente, devono utilizzare le credenziali estratte dal processo di creazione dinamica tramite IaC.

Il sopra citato tf-destroy (12a) è un passo rilevante nello scenario di creazione dinamica degli ambienti il cui scopo è quello di distruggere l'infrastruttura nel momento in cui terminano i test sull'ambiente di review. Al momento della scrittura, in Gitlab Community Edition, non è possibile reagire all'evento di chiusura di una merge-request; perciò si struttura il processo con la direttiva *when: manual* che consente di attendere una conferma manuale dell'utente.

5.4 Gestione dei segreti

Con segreti si intendono tutti quei valori che, in caso di esposizione verso soggetti malintenzionati, possono causare violazioni di sicurezza. Includiamo in questa categoria le credenziali di accesso a servizi, indirizzi, token, eccetera.

Idealmente nessun segreto è contenuto direttamente nei sorgenti o codificato nella pipeline. Il sistema redazionale non fa eccezione e infatti è stato progettato per recuperare queste informazioni nel momento dell'esecuzione. In Sezione 4.8.2 si è già visto come propagare ai container i segreti per l'accesso all'infrastruttura generata da IaC. Vedremo ora come governare i segreti necessari per la corretta esecuzione della pipeline.

5.4.1 Segreti statici

Sono segreti conosciuti staticamente prima dell'esecuzione della pipeline. È possibile configurarli da una schermata apposita di Gitlab e vengono trasmessi come variabili d'ambiente a tutti i processi che compongono la pipeline. Identifichiamo come segreto statico:

- il certificato SSL per l'esposizione del sisred sul dominio *maggiolicloud.it*
- le credenziali per accedere al registro docker aziendale
- le credenziali per l'accesso al server sonarqube per l'analisi statica del codice
- le credenziali del cloud provider utilizzate dello strumento di IaC

5.4.2 Segreti dinamici

Sono segreti ricavabili dinamicamente nel momento dell'esecuzione della pipeline e dipendono sia dall'evento scatenante che dal ramo di origine dell'evento (che determina l'ambiente di rilascio). Gran parte dei segreti dinamici è valorizzato in automatico da Gitlab ed è inserito all'interno dei processi della pipeline tramite variabili d'ambiente. Tuttavia esiste una categoria di segreti dinamici che non ricade in questa gestione. Stiamo parlando dei segreti inerenti le credenziali di accesso per accedere all'infrastruttura creata automaticamente tramite IaC; ad esempio:

- le credenziali del cluster kubernetes utilizzate per rilasciare i singoli microservizi.
- url dell'ambiente su cui lanciare i test e2e.

Senza questi segreti è impossibile completare tutti il ciclo di rilascio e il loro valore non può che essere estratto durante l'esecuzione della pipeline. La soluzione è stata implementata estraendo questi valori dall'output di terraform e introducendo l'uso della specifica **dotenv**[2], ossia un file testuale contenente, in formato chiave valore, dei segreti calcolati a runtime e formalizzato alla fine del processo di iac-apply (8a) leggendo i valori dallo stato di Terraform. Gitlab supporta nativamente questa specifica e, una volta individuato il file dotenv, estrarrà i valori contenuti all'interno e li considererà normali segreti da propagare verso i successivi processi della pipeline.

[2] `https://github.com/motdotla/dotenv`

```
tf-apply:
  stage: iac:apply
  image: "hashicorp/terraform:0.14.5"
  variables:
    TF_KUBE_CONFIG: "$CI_PROJECT_DIR/kubeconfig"
  script:
    - cd $TERRAFORM_DIRECTORY
    - terraform apply -auto-approve $PLAN_FILE
    - ../../export-outputs.sh # output to deploy.env
  artifacts:
    reports:
      dotenv: $TERRAFORM_DIRECTORY/deploy.env
  rules:
    - if: $CI_MERGE_REQUEST_SOURCE_BRANCH_NAME =~ /^review-.*$/ || $CI_COMMIT_BRANCH
      =~ /^(test|master)$/
```

Listato 5: Propagazione IaC outputs tramite dotenv file

5.5 Ottimizzazioni

L'esecuzione di un numero di processi così articolata ha spinto il team del sistema redazionale a dover studiare delle tecniche di ottimizzazione che potessero ridurre considerevolmente la durata del ciclo di rilascio.

5.5.1 Organizzazione per microservizio

L'organizzazione a mono-repository fa sì che ogni modifica effettuata alla base di codice provochi una esecuzione della pipeline per tutti i microservizi del repository. Questo, oltre a violare il principio di rilascio indipendente dei microservizi, investe una spropositata quantità di tempo in cicli di rilascio totalmente non necessari.

La cosa più immediata, in questo caso, è quella di individuare i processi della pipeline che sono specifici di un microservizio e, utilizzando la direttiva *rules*, specificarne l'avvio solo quando i commit che hanno fatto scatenare il processo di CI/CD contengono modifiche alla sotto-cartella che gestisce il microservizio. Si veda ad esempio la riga 9 del listato 6.

```
api-build:
  image: maven:3-adoptopenjdk-14
  stage: build
  script:
    - cd api
    - mvn -ntp compile -Dmaven.repo.local=$MAVEN_USER_HOME
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event" || $CI_COMMIT_BRANCH =~
    ↪  /^(dev|test|master)$/
      changes: ["*", "api/**/*"]
```

Listato 6: Compilazione dell'api gateway nella pipeline

Per quanto riguarda i microservizi, ora che si è sistemata la parte legata alla performance, si può procedere a quella della manutenibilità. Avere infatti un unico file *.gitlab-ci.yml* nella radice del repository risulta complesso e poco leggibile. Si decide di isolare i processi dell'intera pipeline in sotto file che rappresentano i confini logici dei microservizi. Di conse-

guenza i file saranno fisicamente collocati nelle cartelle dei vari microservizi e conterranno solamente i job che compongono il ciclo di vita del microservizio che li ospita.

Questo tipo di organizzazione risulta possibile grazie al meccanismo **include** di Gitlab CI/CD. Il file *.gitlab-ci.yml* nella radice del repository non farà altro che dichiarare variabili globali, gli stage, eventuali definizioni di processi generici (da cui i processi dei microservizi possono estendere) e tramite *include* referenziare tutti i frammenti di .gitlab-ci.yml presenti nelle sottocartelle dei microservizi. Scatenata una pipeline Gitlab si preoccuperà di effettuare l'unione di tutti i frammenti in un unico file consistente e poi proseguirà seguendo il solito comportamento.

5.5.2 Cache

Un'altra tematica fondamentale riguarda il tempo investito nel ripetere le stesse identiche operazioni. L'esempio più calzante di questo tema riguarda il recupero delle dipendenze ai fini della compilazione. Sia Maven che Yarn impiegano diversi minuti per completare questa operazione e la maggior parte delle volte le specifiche delle dipendenze non variano nemmeno tra una pipeline e l'altra. Per preservare il lavoro e riutilizzarlo nei processi di altre pipeline, Gitlab mette a disposizione la direttiva **cache**. Basterà indicare quale percorso contiene i file da preservare e che politica utilizzare: pull-push vs pull. Pull-Push utilizza la cache e la aggiorna con eventuali nuovi file generati durante l'esecuzione del processo. Pull invece utilizza la cache senza aggiornarla. Quest'ultima politica viene utilizzata quando la cache da aggiornare è molto ampia, evitandone il trasferimento in entrambe le direzioni.

```
1  web-build:
2    stage: build
3    image: node:15
4    script:
5      - cd web
6      - yarn install
7    cache:
8      key: "$CI_COMMIT_REF_SLUG-web-cache"
9      paths:
10       - web/node_modules
11     policy: pull-push
12   rules:
13     - if: $CI_PIPELINE_SOURCE == "merge_request_event" || $CI_COMMIT_BRANCH =~
       ↪  /^(dev|test|master)$/
14       changes: ["*", "web/**/*"]
```

Listato 7: Esempio di utilizzo della cache nella compilazione del microservizio web

Capitolo 6

Analisi dei risultati ottenuti

6.1 Introduzione

Nel capitolo precedente abbiamo esposto il risultato di tutta la progettazione dell'automazione. Vogliamo ora analizzare in modo oggettivo se tale sistema introduce o meno vantaggi al ciclo di vita del software. Verranno prese in considerazione le stesse metriche esposte durante l'analisi del sistema originale (si veda Sezione 2.4), pertanto non si riporteranno le relative definizioni.

In questo caso, nel leggere le metriche, si tenga conto che i nuovi valori si riferiscono a medie calcolate sul periodo di attività del sistema: da Luglio 2020 a Febbraio 2021.

6.2 Estrazione delle nuove metriche

Di seguito una spiegazione di come sono state estratte le nuove metriche e un breve commento sui nuovi valori. Nella tabella 6.1 un sommario dell'impatto dell'adozione di architettura microservizi e pratiche DevOps rispetto all'applicazione originale.

Metrica	**Unità**	**Prec.**	**Oggi**	**Cambiamento**
Frequenza di rilascio	$^{\text{rilasci}}/_{\text{giorno}}$	0.071	2.7	+3700%
Durata ciclo di rilascio	ore	8/24	0.19	-97.6%/-99.2%
Commits per giorno	$^{\text{commit}}/_{\text{giorno}}$	2	7.1	+255%
Tempo medio di recupero (MTTR)	ore	36	0.5	-98.6%
Tickets di supporto aperti	$^{\text{tickets}}/_{\text{mese}}$	40	19	-52.5%
Tempo risoluzione problematiche	giorni	4	3	-25%
Interruzione di servizio	$^{\text{minuti}}/_{\text{notte}}$	30	0	-100%
Preparazione ambiente di sviluppo	minuti	120	9	-92.5%
Creazione ambiente di produzione	ore	16	0.35	-97.8%

Tabella 6.1: Sommario delle metriche considerate per valutare l'impatto dell'azione dell'approccio DevOps su un'architettura a microservizi.

Frequenza dei rilasci in produzione

- Valore originale: 0,071 rilasci al giorno (1 volta ogni 14 giorni)
- Valore attuale: 2,7 rilasci al giorno
- Fonte: dato misurato da frequenza pipeline CI/CD ramo master

L'operazione, oltre a essere promossa dalla pratica DevOps di integrazione continua, ora è totalmente automatizzata. A differenza del modello originale dove l'operazione di rilascio allocava uno sviluppatore per un'intera giornata, ora quest'ultimo può dedicarsi ad altro mentre il sistema procede in automatico. Inoltre nel modello a microservizi i rilasci impattano solo quelli che sono effettivamente cambiati, limitando così l'impatto di gestione sull'intero ciclo di rilascio.

Durata ciclo di rilascio

- Valore medio: 1 giorno
- Valore attuale: circa 11 minuti
- Fonte: dato ricavato dalla durata media delle pipeline su branch master

Si passa da un modello manuale con scambio di artefatti effettuato tramite email a un sistema di rilasci totalmente automatico che non prevede interazione umana.

Tempo di sviluppo di un'idea

- Valore originale: 14 giorni
- Valore attuale: 3,7 giorni
- Fonte: Gitlab Merge Request Analytics (da feature branch a dev)

Seppur le abilità individuali dei membri del team rimangano costanti, questo dato si può giustificare rispetto due pratiche introdotte con l'approccio DevOps: *(i)* effettuare CI riduce notevolmente i costi di integrazione rispetto all'integrazione totale a fine sviluppo funzionalità; *(ii)* i controlli di qualità sono effettuati automaticamente durante tutta la fase di sviluppo (durante CI) e nel caso di creazione di ambiente feature review un addetto al controllo della qualità può seguire gli sviluppi e dare feedback costante agli sviluppatori (in contrapposizione al modello originale dove a fine sviluppo veniva richiesto ai sistemisti di installare tali modifiche in ambiente stage).

Numero di commit al giorno

- Valore originale: 2
- Valore attuale: 7.1
- Fonte: Gitlab analytics (branch develop)

DevOps promuove l'integrazione continua del lavoro di tutto gli sviluppatori. Nel modello originale il sistema di versionamento non aveva supporto al tracciamento delle problematiche e alcuna automazione: gli sviluppatori lo utilizzavano come sistema di backup del codice e pertanto usavano effettuare commit a fine giornata.

Tempo medio di recupero (MTTR)

- Valore originale: 1,5 giorni
- Valore attuale: circa 30 minuti
- Fonte: dato misurato

In uno scenario di rilascio su ambiente Cloud, come quello del nuovo sistema redazionale, il motivo più probabile di guasto grave è dovuto a corruzione dovuta a bug nel software, piuttosto che fallimento critico dell'hardware (es: migrazione del database fallita). Di conseguenza viene naturale ricorrere a funzionalità di ripristino temporale offerti dagli stessi servizi cloud (es: ripristinare il database alle 9:30 di questa mattina). Anche scenari più disastrosi che prevedono la rigenerazione totale di un ambiente produzione si risolvono con operazioni per lo più automatiche (vedi metriche *Tempo per la creazione di un ambiente di produzione* e *Tempo di Rilascio*).

Tasso di fuga degli errori

- Valore originale: al 50% dei rilasci succede almeno una commit di hotfix
- Valore attuale: 10% dei rilasci succede almeno una commit di hotfix
- Fonte: Analisi su branch denominate "hotfix-"

Rispetto alla metodologia originale, ai controlli di qualità manuali svolti sull'ambiente di test, si affiancano livelli di controllo automatici che intercettano la maggior parte degli errori: unit test, analisi statica del codice e analisi delle vulnerabilità dei container e test e2e sull'intera applicazione.

Numero di issue/ticket aperti

- Valore originale: 40 issue al mese
- Valore attuale: 19 issue al mese
- Fonte: Gitlab Issue Analytics

Il valore attuale risulta dimezzato. Un fattore chiave che giustifica questo dato, oltre ai diversi controlli di qualità introdotti nel ciclo di sviluppo, è l'introduzione di un sistema di monitoraggio che ci permette di essere proattivi rispetto alle anomalie rilevate sull'ambiente di produzione. Questo permette di risolvere eventuali problemi ancor prima che l'utente possa rilevarli ed aprire un ticket.

Tempo di risoluzione delle problematiche

- Valore originale: 4 giorni
- Valore attuale: 3 giorni
- Fonte: dato misurato su portale Gitlab

Il valore rimane pressoché lo stesso. La spiegazione è data dal fatto che questa metrica cattura l'abilità del team di risolvere problematiche legate al codice. Questa abilità, seppur influenzabile da strumenti che migliorano la qualità di sviluppo dell'applicazione, rimane legata alle singole capacità degli sviluppatori del gruppo di sviluppo.

Tempo di inattività del sistema

- Valore originale: circa 30 minuti ogni notte
- Valore attuale: nessuno
- Fonte: dato misurato

Nella nuova implementazione il sistema non ha momenti di interruzione pianificati. Gli aggiornamenti vengono eseguito *a caldo* e tramite tecnica di rolling update non c'è mai interruzione di servizio.

Tempo impiegato configurazione ambiente sviluppo

- Valore originale: 2 ore
- Valore attuale: circa 9 minuti
- Fonte: dato misurato da esecuzione *docker-compose up*

Per avere un ambiente di sviluppo i passi da effettuare sono due: *git clone* del repository e docker-compose up. Misuriamo il docker-compose up di una macchina che non ha in cache nessuna immagine pre-esistente. NB: il dato è influenzato dalla velocità di download.

Tempo per la creazione di un ambiente di produzione

- Valore originale: 2 giorni
- Valore attuale: circa 21 minuti
- Fonte: dato ricavato da durata pipeline su ambiente review

Si misura la creazione dell'ambiente dinamico tramite pipeline di review, che include creazione infrastruttura tramite terraform, rilascio dell'intero stack applicativo e validazione del sistema tramite test e2e.

6.3 Altre metriche e statistiche

Queste metriche sono abilitate dal nuovo flusso di lavoro e non tutte dipendono dal grado di automazione ma piuttosto dal flusso di lavoro utilizzato. Saranno riportate in quanto forniscono una misura sull'efficienza del metodo di lavoro e possono essere usate come base comparativa per eventuali evoluzioni future del metodo di lavoro.

Distribuzione dei commit

Grafico per distribuzione giorno del mese e orario del giorno. L'andamento omogeneo dei commit valida un utilizzo corretto della pratica di integrazione continua.

- Fonte: Gitlab Value Stream Dashboard

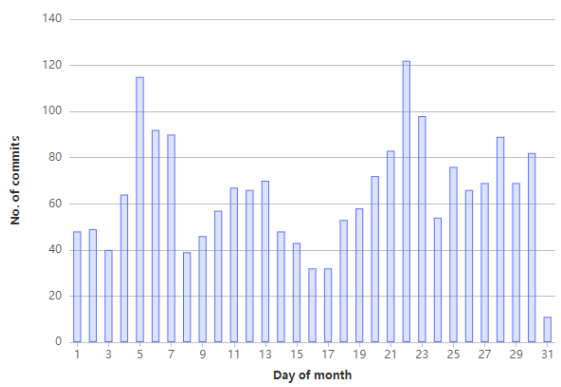

Figura 6.1: Commit per giorno del mese

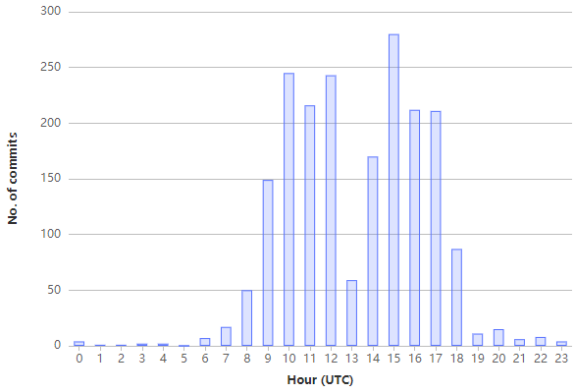

Figura 6.2: Commit per ora del giorno

Pipeline completate con successo

Relazione tra pipeline eseguite e quelle completate con successo. Nel leggere il grafico si consideri che una pipeline che non fallisce mai può essere considerata carente dal punto di vista dei controlli messi in pratica, con conseguente rilascio di prodotti di qualità inferiore. Dall'altro lato si può intendere di successo il sistema di controllo messo in atto nell'ambiente di sviluppo, intercettando i problemi ancor prima di pubblicare il commit.

- Fonte: Gitlab Value Stream Dashboard

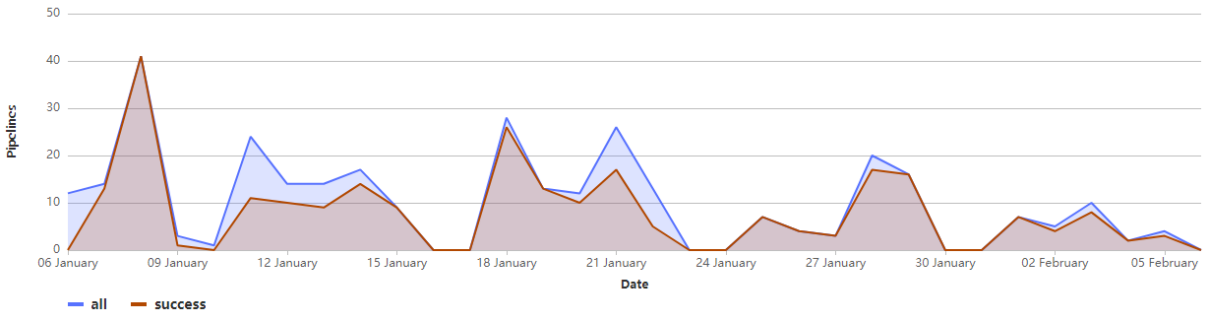

Figura 6.3: Tasso di successo pipeline Gennaio 2021

Tasso di accettazione delle Merge Request

Indica il numero di merge request integrate per mese.

- Valore medio: 31.5 (Max: 47)
- Fonte: Gitlab Value Stream Dashboard

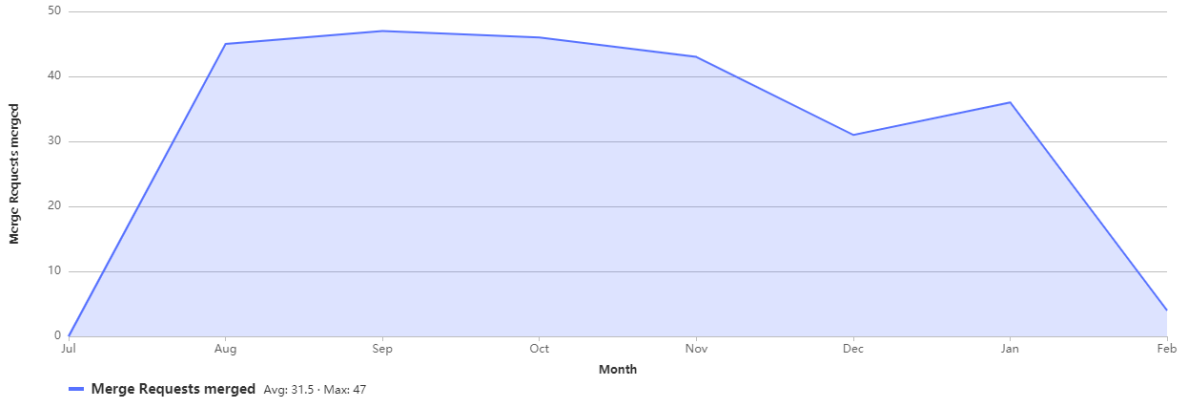

Figura 6.4: Tasso di accettazione delle Merge Request 07/2020 - 02/2021

Durata della fase di code review

Tempo trascorso dalla richiesta di merge request a relativa accettazione.

- Valore: circa 3 ore
- Fonte: Gitlab Value Stream Dashboard

Tempo di esecuzione dei test automatici

Tempo impiegato dalla pipeline per eseguire processi dedicati ai test (unitari, analisi statica, vulnerabilità ed e2e).

- Valore: 4 minuti
- Fonte: Gitlab Analytics

6.4 Analisi dell'impatto

Analizzando i dati raccolti in questo capitolo, la prima ovvia lezione appresa è che l'adozione di DevOps e la trasformazioni di applicazioni monolitiche in architetture a microservizi possono portare a risultati estremamente vantaggiosi. Applicando un confronto sulla totalità degli indicatori di performance non si è mai in una casistica di regressione; in alcuni casi il cambiamento è così d'impatto da essere espresso tramite un altro ordine di grandezza.

Aggregando le valutazioni già riportate metrica per metrica, i vantaggi derivati da un approccio DevOps allo sviluppo sono più evidenti per quelle attività in cui la comunicazione e l'interazione sono un collo di bottiglia. Il modello architetturale a microservizi permette di effettuare rilasci agili, frequenti e senza interruzione di servizio, incrementando la qualità percepita dal cliente. L'automazione dell'infrastruttura non è più un tassello facoltativo dei processi del ciclo di sviluppo del software; di fatto è una risposta efficace al continuo aumento della complessità delle infrastrutture moderne e distribuite. Nel sisred l'approccio di creazione di un ambiente di review dinamico abilita nuove tecniche di controllo della qualità non percorribili nei processi originali e che consentono al programmatore di avere feedback sia sul software che sta sviluppando sia sull'infrastruttura che dovrà supportarlo.

In seguito al lavoro complessivo, svolto nel contesto del sistema redazionale, si ricavano anche lezioni che non sono direttamente estraibili dall'analisi delle metriche.

Innanzitutto la conversione dell'architettura in microservizi richiede che il dominio sia partizionabile in contesti ben isolati mentre per quanto riguarda le pratiche DevOps non esiste uno strumento che riesca a cogliere tutte le sfaccettature legate alla realizzazione automatizzata del ciclo di rilascio; si è visto come il risultato della fase di progettazione non sia altro che una coordinata sequenza di strumenti e tecnologie, orchestrata dalla pipeline di CI/CD. Un'altra lezione consiste nella capacità di saper gestire e allocare il tempo e le risorse necessarie alla manutenzione dei processi e relativi strumenti di automazione, che costituiscono la parte fondante delle stesse pratiche "continue" promosse da DevOps. Le tecnologie, i requisiti aziendali e le necessità di un team evolvono nel tempo e, di conseguenza, anche i processi automatici che li supportano devono evolversi. Infine risulta fondamentale non sottovalutare l'elevata quantità di competenze richieste per effettuare la progettazione, implementazione e coordinamento di un processo di sviluppo del software automatizzato. Rompendo le barriere che separano le figure Dev e Ops il risultato spesso consiste in una figura professionale che deve conoscere entrambi i mondi con lo scopo di operare correttamente e trasversalmente su tutti gli elementi che compongono il ciclo di vita del software. In un contesto aziendale questo non è sempre possibile e spesso determinato da fattori che cambiano da gruppo a gruppo.

In conclusione, seppur ampiamente soddisfatti dei risultati raggiunti dalla trasformazione del sistema redazionale, consigliamo di interpretare i risultati delle nuove metriche come il frutto di una coordinata volontà di cambiare l'approccio allo sviluppo del software. I benefici derivati da DevOps e microservizi introducono infatti una grandissima trasformazione dei processi che, se non sostenuta da un forte cambio di mentalità di tutti i componenti del team, rischia di comportare un drastico calo nella produttività e nella qualità del prodotto finale.

6.5 Sviluppi futuri

Il lavoro svolto nel contesto del sistema redazionale, per quanto abbia raggiunto risultati molto soddisfacenti, lascia aperte possibilità di miglioramento. Il primo consiste nell'implementazione della già citata soluzione ibrida nel modello di gestione dei sorgenti, rimpiazzando il modello a mono-repository in quello gestito con i sotto-moduli. Questa gestione permetterebbe di avere la totale indipendenza nello sviluppo dei microservizi.

Un'altra tematica molto importante, sopratutto in ottica DevOps, riguarda l'adozione di strategie di rilascio che possano mettere in pratica i concetti studiati in Sezione 1.5.6, consentendo di propagare gli aggiornamenti verso una base di utenti ristretta, analizzare i dati raccolti su questo campione della popolazione e correggere eventuali anomalie prima del rilascio completo. In ottica futura, implementare questo processo, unito alla già avanzata gestione di IaC e ambienti di rilascio, potrebbe giustificare l'introduzione di una piattaforma specifica per il continuos deployment che subentra subito dopo il rilascio degli artefatti dei microservizi da parte di Gitlab.

Infine un miglioramento molto rilevante nel contesto della sicurezza informatica: l'introduzione di un meccanismo per la gestione dei dati sensibili e dei segreti. Un servizio di questo tipo abiliterebbe nel nostro processo la politica di rotazione dei segreti per l'accesso

all'infrastruttura e i servizi, ovvero un meccanismo periodico e automatico di rinnovo delle credenziali di accesso. Limitare la durata di una password riduce il rischio e l'efficacia di attacchi basati su *brute-force* ed errori umani (si pensi un erroneo versionamento della password nel codice sorgente) condensando la finestra di tempo durante la quale una password rubata potrebbe essere ancora valida. L'introduzione di questa pratica è fortemente agevolata dal già presente meccanismo di IaC e, una volta tradotta in codice, potrebbe lavorare autonomamente durante le pipeline, incapsulando tutta la logica di rotazione e rimuovendola dagli oneri degli sviluppatori.

Capitolo 7

Conclusioni

In questa tesi si è documentato il processo che ha portato un gruppo ristretto di sviluppatori di una grande azienda del territorio a riprogettare un prodotto software monolitico e il relativo processo di sviluppo attraverso la trasformazione architetturale a microservizi e l'adozione della filosofia DevOps.

Dopo aver effettuato una panoramica sullo stato dell'arte di princìpi, pratiche e strumenti della filosofia DevOps, si è proceduto all'analisi della situazione attuale del software da riprogettare: tramite un'intervista ai membri del team è stato ricavato un quadro della situazione originale in termini di pratiche, architettura e una serie di indicatori di prestazione che potessero esprimere in maniera oggettiva le metriche associate ai processi in uso. Successivamente si è effettuata la raccolta dei requisiti e dei risultati ottenuti dal ripensamento dell'applicazione monolitica, individuando nella decomposizione a microservizi l'architettura che potesse rispondere alle necessità emerse. Si sono quindi discusse le modifiche architetturali introdotte per scomporre il sistema, il flusso di lavoro DevOps progettato per migliorare il processo di sviluppo del software, la politica di branching associata al controllo di versione e la sua relazione con gli ambienti di esecuzione, la pipeline automatizzata a supporto del processo di rilascio che non richiedesse alcun intervento umano, e infine i diversi controlli qualitativi automatizzati e applicati durante varie fasi di tale pipeline. Infine si è proceduto con il confronto tra le metriche associate ai processi di sviluppo del sistema originale con quello rinnovato, traendone informazioni molto interessanti. Nessuna metrica è regredita e, analizzando i risultati, risulta innegabile che l'utilizzo di pratiche DevOps ha un enorme potenziale per migliorare il processo di produzione del software; sopratutto in scenari dove i princìpi alla base della trasformazione del processo sono supportati dagli stessi che governano la nuova gestione architetturale.

Tra le lezioni apprese vi è stata quella di rendere la transizione uno *sforzo coordinato* condiviso con la totalità dei componenti del team. Ragionare seguendo la filosofia DevOps richiede che tutti i membri del team condividano la stessa linea di pensiero, pena l'introduzione di ulteriore tempo dedicato alla invalidazione e gestione dei processi automatici. Il caso di studio presentato ha ampiamente beneficiato delle ridotte dimensioni del team di sviluppo, unito alla volontà di rinnovare pratiche che erano percepite come ripetitive, macchinose, obsolete e sopratutto dispendiose in termini di tempo. Team più grandi verosimilmente includeranno un contesto con personale e situazioni che renderanno meno efficace l'introduzione del cambiamento: in questi casi, crediamo che la chiave sia *comu-*

nicare e far conoscere i potenziali benefici a tutti i membri, eventualmente confrontandosi con altri casi di successo presenti in azienda.

Sulla stessa linea, esiste un delicato equilibrio tra i benefici derivati dall'adozione di una pratica all'avanguardia e il relativo *costo* della sua introduzione e applicazione, che include sia la curva di apprendimento che l'adattamento a nuove modalità di lavoro. A volte, scegliere una soluzione più semplice, anche se tecnicamente non ottimale, può produrre gli stessi benefici a un costo molto minore. Nella nostra esperienza questo caso si è presentato in più occasioni. Un esempio è stato la scelta della configurazione del repository, dove si è privilegiato un modello a monorepository piuttosto che un repository per microservizio usando i sottomoduli git. Oppure nella scelta della politica di assegnazione delle versioni, dove si è prediletta la totale automazione dell'assegnazione rispetto all'espressività data da un modello manuale come semver.

L'*anticipazione del cambiamento* è un principio consolidato dell'ingegneria del software e crediamo si possa applicare anche al processo legato al ciclo di sviluppo. In particolare tramite l'ampia conoscenza delle tecniche esistenti e sperimentando sugli strumenti attuali, riusciamo a costruire un quadro consapevole di ciò che è fattibile oggi, stimando i costi d'implementazione di una pratica e prevedendo come si evolveranno le tecniche e gli strumenti nel futuro. Questo potrebbe essere sfruttato per ottenere due risultati: in primo luogo, raggiungere il suddetto equilibrio tra benefici, complessità e costo; in secondo luogo, essere pronti ad aggiornare e migliorare le pratiche in uso per reagire efficacemente ai cambiamenti nella struttura di sviluppo (ad esempio, l'acquisizione di nuovi membri del team).

Bibliografia

[1] Johan Abildskov. Additional git features. In *Practical Git*, pages 139–161. Apress, 2020. 50

[2] S.A.I.B.S. Arachchi and Indika Perera. Continuous integration and continuous delivery pipeline automation for agile software project management. In *2018 Moratuwa Engineering Research Conference (MERCon)*, pages 156–161. IEEE, May 2018. 14

[3] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. DevOps: Introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, May 2017. 18

[4] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. DevOps: Introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498. IEEE, May 2017. 18

[5] Xiaoying Bai, W.T. Tsai, R. Paul, Techeng Shen, and Bing Li. Distributed end-to-end testing management. In *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*. IEEE Comput. Soc. 54

[6] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 1st edition, 2015. 18

[7] Yevgeniy Brikman. *Terraform: Up and Running Writing Infrastructure as Code*. O'Reilly Media, Inc., 1st edition, 2017. 67, 68, 73

[8] Brendan Burns. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. O'Reilly Media, Inc., 1st edition, 2018. 63

[9] Pedro Henrique de Andrade Gomes, Rogerio Eduardo Garcia, Gabriel Spadon, Danilo Medeiros Eler, Celso Olivete, and Ronaldo Celso Messias Correia. Teaching software quality via source code inspection tool. In *2017 IEEE Frontiers in Education Conference (FIE)*. IEEE, October 2017. 53

[10] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016. 73

[11] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley, Boston, 2004. 36

[12] Nicole Forsgren. 2015 state of devops report, 2015. 9

[13] Aymeric Hemon-Hildgen, Frantz Rowe, and Laetitia Monnier-Senicourt. Orchestrating automation and sharing in devops teams: a revelatory case of job satisfaction factors, risk and work conditions. *European Journal of Information Systems*, 29(5):474–499, 2020. 10

[14] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 1st edition, 2017. 63

[15] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010. 7

[16] Jez Humble and Joanne Molesky. Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal*, 24(8):6, 2011. 8

[17] Kasun Indrasiri and Prabath Siriwardena. Service mesh. In *Microservices for the Enterprise*, pages 263–292. Apress, 2018. 75

[18] Ciera Jaspan, Matthew Jorde, Andrea Knight, Caitlin Sadowski, Edward K. Smith, Collin Winter, and Emerson Murphy-Hill. Advantages and disadvantages of a monolithic repository: A case study at google. ICSE-SEIP '18, page 225–234, New York, NY, USA, 2018. Association for Computing Machinery. 49

[19] Morgan B. Kamuto and Josef J. Langerman. Factors inhibiting the adoption of DevOps in large organisations: South african context. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pages 48–51. IEEE, May 2017. 10

[20] John Klein and Douglas Reynolds. Infrastructure as code–final report. 2018. 67

[21] Vasanth K. Makam. Continuous integration on cloud versus on premise: A review of integration tools. *Advances in Computing*, 10(1):10–14, 2020. 79

[22] Mathias Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, May 2014. 12

[23] Martin Rütz. Devops: A systematic literature review. *Seminar paper*, 08 2019. 11

[24] Jay Shah and Dushyant Dubaria. Building modern clouds: Using docker, kubernetes & google cloud platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, January 2019. 38

[25] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017. 15

[26] Daniel Stahl, Torvald Martensson, and Jan Bosch. Continuous practices and devops: beyond the buzz, what does it all mean? In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, September 2017. 7

[27] Manish Virmani. Understanding DevOps & bridging the gap from continuous integration to continuous delivery. In *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, pages 78–82. IEEE, May 2015. 8

[28] Eric Wohlgethan. *SupportingWeb Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue. js.* PhD thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2018. 40