POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Analisi e sviluppo della migrazione di un assistente virtuale: da monolite ai microservizi

Relatore
prof. Marco Torchiano

Candidato
Giacomo Inghilleri

Anno Accademico 2022-2023

Sommario

Negli ultimi anni l'architettura a microservizi è diventata un *pattern* architetturale sempre più scelto dalle aziende che hanno la necessità di modernizzare vecchi *software* monolitici. Questi, infatti, rendono complessa l'introduzione di nuove funzionalità e la gestione del codice preesistente. Inoltre, poiché le tecnologie scelte durante la fase iniziale del progetto non possono essere modificate nel corso della vita del *software*, lo *stack* tecnologico sarà fisso e, con il passare degli anni, anche obsoleto. L'idea di svolgere questa tesi nasce dalla necessità dell'azienda Cloudmind S.p.A. di ampliare un *software* monolite al fine di renderlo un assistente virtuale. L'obiettivo è quello di dimostrare che i microservizi rappresentano una valida alternativa all'utilizzo dell'architettura monolitica permettendo il superamento delle problematiche sopracitate.

Indice

Elenco delle figure

1. Introduzione

1.1 L'azienda: Cloudmind S.p.A.

L'azienda in cui è stato svolto questo progetto di tesi si chiama Cloudmind S.p.A. Si tratta di una *startup* innovativa fondata nel 2019 con il nome di Cloudtec. L'azienda ha due sedi, una a Modena e una a Palermo, ed è specializzata nella *Cloud Transformation*, cioè di progettare e implementare sistemi utilizzando le moderne tecnologie *cloud-based*. In particolare, l'azienda è specializzata nei seguenti ambiti:

- **Cloud Collaboration**: produce sistemi per agevolare la collaborazione aziendale con l'obiettivo di semplificare e ottimizzare i processi aziendali, migliorando l'efficienza operativa e promuovendo una maggiore produttività.
- **Artificial Intelligence (AI)**: grazie all'utilizzo dell'intelligenza artificiale sono stati sviluppati assistenti digitali e soluzioni di realtà assistita intelligente. Entrambi si basano sull'elaborazione dei dati e sull'apprendimento automatico per migliorare le loro capacità di riconoscimento. Utilizzano le tecniche di intelligenza artificiale per analizzare grandi quantità di dati, riconoscere modelli e migliorare le prestazioni nel tempo.
- **Servizi DevOps**: adotta la metodologia DevOps, approccio collaborativo che integra lo sviluppo *software* (*Development*) con le operazioni IT (*Operations*), per la realizzazione delle proprie soluzioni e dei servizi offerti.
- **Cloud NetSecOps**: vengono realizzate infrastrutture di rete flessibili e scalabili e protegge le risorse aziendali sfruttando i principi NetSecOps (*Network Security Operations*), metodologia che combina i principi e le pratiche di DevOps con la sicurezza delle reti per realizzare infrastrutture di rete flessibili, scalabili e sicure.

Il suo principale *partner* è Google: questa *partnership* strategica consente a Cloudmind di sfruttare l'ampia gamma di servizi e risorse fornite da Google Cloud per offrire soluzioni di alta qualità ai propri clienti. Google Cloud offre un insieme completo di servizi di *Cloud Computing* che consentono alle aziende di sviluppare, gestire e distribuire applicazioni e servizi *software* sfruttando l'infrastruttura scalabile e affidabile fornita dalla potenza di *Internet*. I servizi di Google Cloud comprendono, tra gli altri, la gestione dei *server*, l'archiviazione dei dati, l'elaborazione dei dati, il *machine learning*, il *data analytics* e la sicurezza.

Nel settembre 2022 l'azienda è entrata a far parte del gruppo Maticmind, una rinomata e consolidata azienda informatica italiana. Dal precedente nome, Cloudtec, l'azienda dopo la fusione ha preso il nome di Cloudmind.

1.2 Il progetto

Questa tesi si pone come obiettivo quello di presentare l'architettura a microservizi come possibile approccio per modernizzare un *software* in versione monolitica. L'intero progetto parte dall'analisi di un applicativo già in uso dai dipendenti dell'azienda che permette loro di affidargli alcune operazioni legate ai processi aziendali.

In particolare, l'idea nasce quando è nata la necessità di ampliare il *software* con nuove funzionalità con l'obiettivo di renderlo un *chatbot* intelligente in grado di conversare con i dipendenti per sostituire l'intervento umano nella configurazione dei progetti aziendali salvati all'interno di *repository* nel Cloud, automatizzando operazioni legate al processo di sviluppo. Inizialmente era stato pensato per eseguire semplici operazioni, per questo si era scelta l'architettura monolitica. Tuttavia, questa lo avrebbe reso nel tempo ingestibile per i seguenti motivi:

- L'introduzione di nuove implementazioni e la gestione delle regressioni diventano più complesse a causa del cattivo isolamento del codice preesistente e dell'assenza di resilienza del *software* stesso.
- Lo *stack* tecnologico scelto per la creazione del *software* è fisso e sarà utilizzato per l'intero ciclo di vita: un eventuale nuovo requisito potrebbe essere

implementato con una tecnologia più efficiente ma questo non è possibile con un monolite.

Per questo motivo infatti è stata scelta l'architettura a microservizi, un recente *pattern* architetturale che risolve alcuni di questi problemi migliorando notevolmente la qualità del *software* da tanti punti di vista.

Nei prossimi capitoli verrà presentato innanzitutto lo stato dell'arte, analizzando l'architettura a microservizi ponendo l'attenzione sui motivi per cui sceglierla sia un'opzione vantaggiosa rispetto alla classica architettura monolitica. Successivamente verranno descritte le tecnologie utilizzate per lo sviluppo della nuova versione del *software*; in parte, sono state utilizzate le stesse tecnologie utilizzate per l'applicativo già esistente. Dopo questa prima fase, verrà presentato il *software* monolitico evidenziando i limiti e motivi per cui si è deciso di migrare verso una nuova architettura. A partire da questo, nel quarto capitolo verrà descritta la nuova versione del *software* sviluppato ponendo l'attenzione anche su come questa scelta implementativa può essere di notevole supporto per i gruppi di lavoro che utilizzano la metodologia *Agile*. Infine, nel quinto capitolo, si effettuerà una valutazione del lavoro svolto, prendendo in considerazione gli obiettivi raggiunti e focalizzandosi su determinati parametri di riferimento.

Il lavoro per la realizzazione di questo progetto ha visto una prima fase di studio. In particolare, ho approfondito gli argomenti relativi ai microservizi e come questi possono migliorare la qualità di un *software* sfruttando il materiale messo a disposizione dall'azienda e dal *tutor* aziendale. Successivamente mi sono dedicato a studiare la documentazione della piattaforma Slack, utilizzata dai dipendenti aziendali per poter interagire con il *software*. In particolare, ho approfondito gli strumenti messi a disposizione da Slack per gli sviluppatori. Dopo questa prima fase di studio, sotto la supervisione del *tutor*, ho esaminato il *software* monolitico da migrare verso la nuova architettura. Per questo, infatti, mi è stato fornito l'accesso alla piattaforma Gitlab aziendale per potere accedere al codice sorgente del progetto. A partire da questo abbiamo definito i microservizi da creare e le tecnologie che ciascuno di essi doveva avere. Dopo aver definito l'intera struttura, mi sono dedicato alla stesura del codice. Per la fase di implementazione sono stati definiti dei *task* da eseguire e per ciascuno di essi il *tutor* ha avuto il ruolo di *Code Reviewer*. In questo modo, con le indicazioni e consigli ricevuti,

ho potuto migliorare il codice scritto utilizzando le *best practice* delle tecnologie utilizzate.

2. Stato dell'arte

2.1 Microservizi

2.1.1 Architettura Monolitica

L'architettura monolitica è il classico approccio allo sviluppo del *software*, utilizzato in passato da grosse aziende come Amazon ed Ebay [1]. La caratteristica principale è quella di avere tutte le funzionalità incapsulate in un'unica applicazione; questo implica che i suoi moduli non possono essere eseguiti e sviluppati in maniera indipendente [2].

La struttura generale del monolite solitamente prevede tre livelli:

- *Presentation Layer*: è il livello più alto di un *software* monolite. In questo livello è presente la *User Interface* (UI), cioè tutto il codice che si occupa di generare l'interfaccia grafica che dà la possibilità all'utente di poter interagire con il resto del *software*. Inoltre, contiene tutta la logica che si occupa di presentare correttamente il codice che arriva dai livelli inferiori.
- *Business Logic Layer*: questo livello racchiude la maggior parte della logica dell'intero *software*. In particolare, contiene tutti quei moduli che si occupano di eseguire le funzionalità dell'applicazione. A questo livello avviene l'elaborazione del dato che deve essere inviato al livello superiore per essere mostrato all'utente oppure deve essere elaborato per essere memorizzato nello *storage* sfruttando il livello inferiore.

- *Data Access Layer*: è il livello più basso di un applicativo e contiene i moduli che interagiscono e gestiscono l'accesso al *database*. Lo scopo di questo livello, quindi, è quello di fornire dei metodi al livello superiore per dare la possibilità di leggere dati dal *database* oppure memorizzarli.

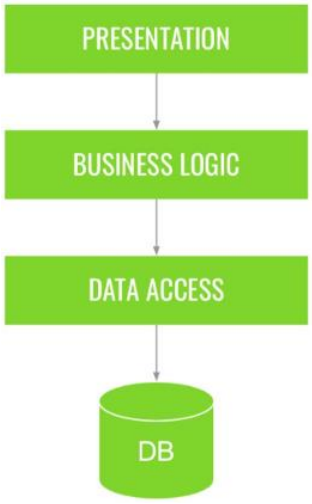


Figura 1 - Modello dell'architettura a strati utilizzato per rappresentare la struttura di un monolite.

Tutti e tre i livelli fanno parte dello stesso modulo. Internamente può essere garantita una suddivisione logica ma fanno parte dello stesso progetto; quindi, devono seguire tutti insieme le fasi di sviluppo. Per questo, infatti, sono in piena sinergia e non occorrono sistemi di comunicazione esterna tra i livelli.

La semplicità data dal poter gestire un unico progetto e la velocità iniziale dello sviluppo spiegano il motivo per il quale la maggior parte dei *software*, creati nel passato, ha seguito questa architettura. Tuttavia, quando il monolite inizia ad essere aggiornato e a crescere aumentando il numero di funzionalità, emergono le debolezze di questo tipo di approccio [1].

In "Microservices Patterns With Examples in Java" [3], Richardson racchiude in sei punti le vere limitazioni di questa architettura, definita come "l'inferno monolitico":

- La complessità intimidisce gli sviluppatori. Più gli sviluppatori si trovano davanti a tante righe di codice, più aumenta il tempo necessario per comprendere la parte del progetto su cui lavorare.

- Lo sviluppo è lento. Dato che per ogni aggiunta di una nuova funzionalità occorre testare l'intera applicazione, il processo di *deployment* diventa oneroso.
- Il processo di consegna è lungo e arduo. Con questo tipo di approccio non si possono eseguire rilasci graduali, tipici dell'approccio *agile*, perché bisogna rilasciare l'intero prodotto funzionante.
- Dimensionamento difficile. I moduli dell'applicazione avendo delle funzionalità diverse spesso richiedono risorse differenti. Per questo il monolite costringe a scegliere una macchina in grado di soddisfare i requisiti dei vari moduli.
- Mantenere un monolite affidabile è molto difficile perché l'intera applicazione è un unico *point of failure*: tutti i moduli dell'applicazione sono eseguiti con lo stesso processo; quindi, se qualche *bug* interrompe il funzionamento di un modulo l'intero sistema non funziona più.
- Il *software* è bloccato nel suo *stack* tecnologico. Quando si inizia il progetto occorre decidere quali tecnologie usare, e queste non possono essere cambiate in corso d'opera, a meno di costi e tempi eccessivi; quindi, man mano che il *software* cresce ci si ritrova a lavorare con tecnologie sempre più obsolete.

Al fine di rendere il *software* analizzato nell'ambito di questa tesi aperto all'aggiunta di nuove funzionalità e flessibile ai cambiamenti, è nata la necessità di trovare un'architettura molto più solida per un prodotto sempre più responsabile, cioè in grado di eseguire operazioni importanti per i processi aziendali.

2.1.2 Architettura a Microservizi

L'architettura a microservizi è un approccio architetturale utilizzato per sviluppare e implementare applicazioni *software* come una *suite* di piccoli servizi granulari che possono essere integrati tramite meccanismi di comunicazione, normalmente API REST, *representational state transfer* [4]. Si tratta quindi di un'architettura *software* strutturata e organizzata in tanti servizi autonomi che lavorano insieme, ciascuno dei quali è focalizzato su una specifica funzionalità, di responsabilità molto limitata [5]. La

peculiarità sta nel fatto che i servizi tra di loro devono essere fortemente disaccoppiati: questo fa sì che i servizi siano autonomamente scalabili.

In ottica di "*dividi et impera*" [6], quindi ciascun microservizio può essere visto come un piccolo *software* che partecipa al funzionamento dell'intero sistema e lo fa tramite interfacce di comunicazione implementate. Generalmente ciascun servizio interagisce con gli altri attori dell'architettura esponendo delle API. Quindi, tramite l'utilizzo di comandi HTTP viene garantito il funzionamento dell'intero sistema.

In figura 2 è raffigurata la struttura generale di un'architettura a microservizi. È importante notare come a differenza di un'architettura monolitica l'intera logica è divisa in N servizi differenti tra di loro. Inoltre, per il principio di disaccoppiamento si prevede che ciascun servizio abbia un database proprio in modo tale da evitare anche la condivisione del dato. Tuttavia, gestire un numero di *database* pari al numero di servizi ha un costo elevato, quindi la pratica comune è quella di tenere un numero minimo *database* comuni ai vari microservizi. Nella figura è rappresentato anche l'API *Gateway*, cioè quel componente architetturale che si pone a metà strada tra il *client* e i servizi in *back-end* ed ha il ruolo di gestire le richieste che vengono effettuate.

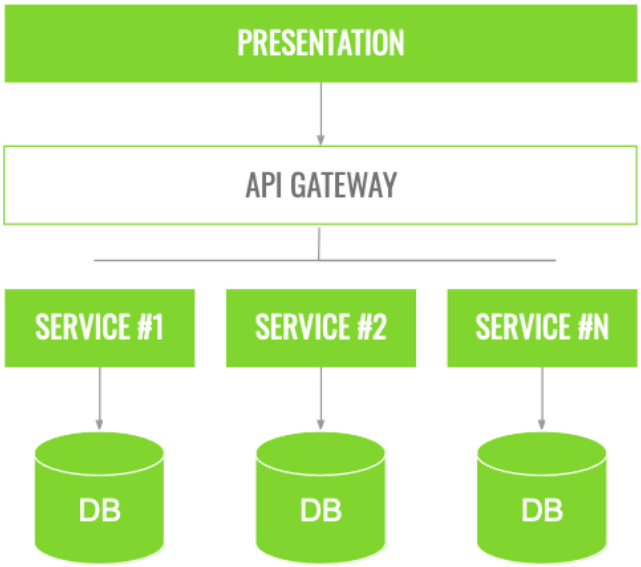


Figura 2 - Struttura generale dell'architettura a microservizi.

Per certi aspetti la logica modulare introdotta e applicata dall'architettura a microservizi non sembra essere una novità. Questo perché l'idea di scomporre l'applicazione in parti sempre più piccole ciascuna delle quali è responsabile di una singola funzionalità ricorda le *Service-Oriented Architecture* (SOA): si tratta di una metodologia di sviluppo molto nota e affermata che, come si vede rappresentato in figura 3, prevede di impostare l'applicazione in tanti servizi che comunicano tra di loro attraverso un'infrastruttura *software* di tipo *Enterprise Service Bus* (ESB), un canale di comunicazione all'interno del quale ciascun servizio invia i messaggi.

Molti studiosi e sostenitori della *Service-Oriented Architecture* sostengono infatti che i microservizi siano soltanto un approccio di implementazione alla SOA. In realtà c'è una differenza molto importante tra le due architetture: le *Service-Oriented Architecture* si basano sull'idea di condividere il più possibile mentre le architetture a microservizi sono costruite per condividere il meno possibile, e quindi rendere ciascun servizio autonomo e indipendente [4]. Questo perché l'*Enterprise Service Bus* utilizzato nelle SOA può diventare sia un collo di bottiglia per la gestione dei messaggi di tutti i servizi sia un *point of failure*: se si verificano dei problemi l'intero sistema ne risente e quindi smetterebbe di funzionare.

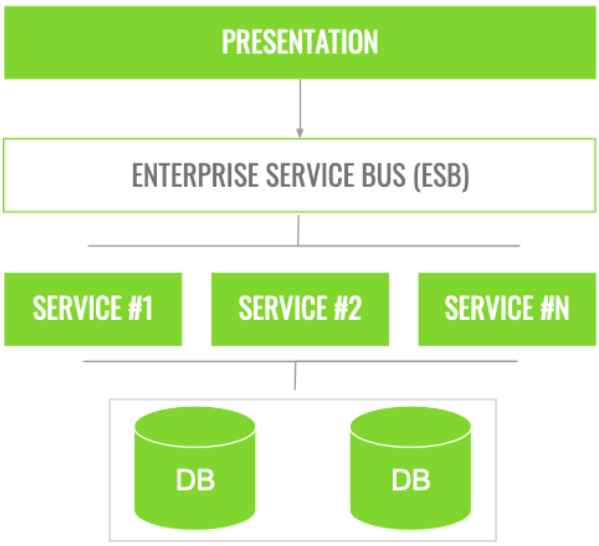


Figura 3 - Struttura dell'architettura Service-Oriented.

Invece, la novità introdotta dai microservizi che li fa distaccare anche dalle SOA e li rende innovativi rispetto al passato sta nel fatto che essi utilizzano un livello di astrazione superiore nella comunicazione tra i vari servizi. Invece di utilizzare un canale comune i microservizi fanno uso di API, il che li rende molto più isolati e indipendenti rispetto all'uso di un *bus* comune. Per questo, infatti, molte aziende al giorno d'oggi scelgono sempre di più di utilizzare i microservizi. Il *software* risulta molto affidabile e robusto e il guasto di una sola parte non intacca il resto del sistema.

Come già descritto prima, i microservizi sono indipendenti l'uno dall'altro. Per questo motivo si utilizzano ambienti virtualizzati per eseguire ciascun servizio rendendo quindi il sistema più sicuro e facile da gestire.

Al giorno d'oggi per virtualizzare un servizio si utilizzano i *container*, cioè pacchetti di *software* leggeri, autonomi ed eseguibili che includono tutto il necessario per eseguire una singola applicazione: codice, strumenti di sistema, librerie di sistema e impostazioni. L'ambiente che viene creato ha delle risorse limitate in base alle operazioni che deve eseguire ed è disaccoppiato dal sistema operativo e dall*'hardware* della macchina in cui viene lanciato [7].

La virtualizzazione dei microservizi ha permesso la veloce crescita degli ambienti *cloud*, come Google Cloud e Amazon Web Services (AWS) che permettono di rilasciare i singoli servizi sul *cloud* garantendo quindi lo sviluppo di sistemi interamente *cloud-native*.

2.1.3 Vantaggi e svantaggi

Per questo progetto è stata scelta questa architettura perché non solo risolve in modo puntuale i problemi presentati nell'introduzione, ma permette anche di rendere il *software* più robusto durante l'intero ciclo di vita.

La suddivisione del codice monoblocco in tanti servizi autonomi e atomici riduce notevolmente la complessità del processo di sviluppo: l'architettura può essere riflesso della suddivisione in *team* degli sviluppatori, assegnando la gestione di un servizio ad ognuno di essi. Infatti, seguendo questo principio, gli sviluppatori dedicano tutte le loro

forze all'implementazione e alla fase di *test* di poche e coesive funzionalità [8] e i *team* diventano autonomi e indipendenti dalle linee di sviluppo degli altri.

In questo modo, è possibile eseguire rilasci più frequenti contenenti piccole modifiche, a differenza della controparte monolitica che richiederebbe rilasci più consistenti con l'obbligo di testare ogni volta l'intero *software*.

Questo approccio al giorno d'oggi è preferito da tante aziende perché agevola l'impiego della metodologia *Agile*, garantendo al cliente consegne frequenti e in tempi brevi a differenza dell'approccio *waterfall* che prevede la consegna solo dopo aver completato e testato l'intero prodotto; questo implica una maggiore flessibilità di fronte alle richieste di modifica da parte del cliente.

Poiché ciascun servizio è indipendente da tutti gli altri non è necessario che ognuno di essi sia implementato con la stessa tecnologia, offrendo quindi la possibilità di sviluppare i servizi utilizzando tecnologie *ad hoc* e secondo necessità. Se ci si ritrova a lavorare con delle tecnologie che negli anni sono diventate obsolete, è possibile riscrivere il servizio: il costo da pagare sarà sicuramente più basso che rifare l'intero progetto. Oppure se una parte di un sistema richiedesse migliori *performance*, si potrebbe decidere di usare uno *stack* tecnologico diverso in grado di raggiungere i livelli richiesti [5].

Un altro motivo per cui si scelgono i microservizi è che questi aumentano notevolmente la tolleranza ai guasti. Newman, in "Building Microservices" [5], definisce un sistema a microservizi "resiliente". Infatti, avere un sistema suddiviso in tanti microservizi, ciascuno responsabile di eseguire determinate parti del *software*, fa sì che il blocco di un singolo servizio non causi l'arresto dell'intero sistema, perché gli altri servizi attivi saranno ancora in grado di svolgere le funzioni di cui sono responsabili, evitando inoltre una possibile caduta a domino grazie all'utilizzo di *circuit breaker* [9].

Nonostante l'architettura a microservizi sia innovativa e risolva i problemi già discussi, non è sempre il *pattern* ideale per l'implementazione di un *software* perché, rispetto all'approccio tradizionale, introduce ulteriore complessità aumentando i costi e le risorse necessarie.

La scomposizione delle funzionalità in microservizi richiede uno sforzo in più da parte dei *software architect* sia perché devono lavorare con più progetti, ciascuno dei quali

segue un percorso di sviluppo differente, sia perché per far funzionare l'intero *software* devono permettere la comunicazione tra i servizi.

Inoltre, è richiesto un maggiore *effort* in fase di *debug* ed una conoscenza più trasversale delle tecnologie.

Per di più, ciascun microservizio ha bisogno della propria infrastruttura per poter funzionare quindi il costo è più alto rispetto a quello che generalmente si ha con l'approccio monolitico. Per questo, infatti, la scelta dell'architettura dipende tanto anche dal costo che il cliente è disposto a pagare.

Implementare un *software* monolita, invece, implica sviluppare una singola applicazione che contiene al suo interno tutti i moduli necessari per poter funzionare, senza la necessità di gestire più servizi; quindi, per ciascuna modifica il processo di sviluppo è centralizzato e non distribuito su più parti. Infatti, implementare e mantenere funzionante un'architettura monolitica è molto più semplice ed economico rispetto alla gestione di un sistema distribuito [10].

Occorre quindi analizzare i requisiti del sistema e chiedersi quanto un applicativo debba essere flessibile nel tempo perché, se è destinato a crescere e ad accogliere nuove funzionalità, c'è il rischio che diventi una enorme monolite difficile da gestire.

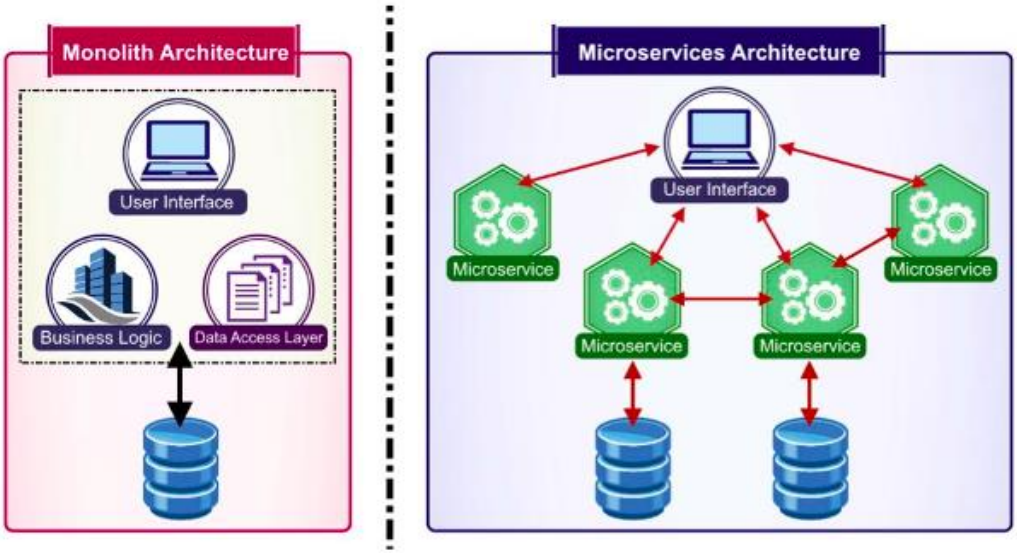


Figura 4 - Rappresentazione di un'architettura monolitica a confronto con un'architettura a microservizi.

2.2 Tecnologie utilizzate

In questo paragrafo sono descritte le tecnologie utilizzate per l'implementazione della nuova versione del *software*. Come già anticipato nell'introduzione, in parte sono tecnologie precedentemente selezionate per l'implementazione del *software* in versione monolitica. Per la realizzazione della nuova versione del *software*, sono state scelte tecnologie per garantire un miglioramento delle prestazioni e delle funzionalità dell'applicazione.

2.2.1 Typescript

Typescript è un linguaggio di programmazione *open source* sviluppato da Microsoft nel 2012 come estensione di Javascript. Può essere definito con un sovrainsieme di Javascript, perché ingloba tutte le sue funzionalità e proprietà aggiungendone alcune proprie.

Javascript è nato come linguaggio di *scripting* da integrare all'interno delle pagine *web*. Con il passare degli anni si è evoluto fino a diventare un linguaggio comune per lo sviluppo di applicazioni *web* e *mobile*, quindi, per progetti molto più grandi rispetto a semplici *script*. Per sua natura, quindi, non si presta bene ad applicativi di grandi dimensioni. Per questo, infatti, Typescript aggiunge alcune funzionalità per offrire un linguaggio potente come Javascript, ma molto più adatto a creare *software* sicuri, robusti e facilmente mantenibili.

In particolare, in Javascript le variabili dichiarate non hanno un tipo unico ma cambia in base al valore che le viene assegnato. Per grandi progetti, la tipizzazione dinamica aumenta notevolmente la probabilità di avere *bug* ed errori. Inoltre, il livello di sicurezza è molto basso poiché non c'è nessuna garanzia sui tipi che le variabili contengono. In Typescript, invece, è stata introdotta la tipizzazione forte delle variabili che fa aumentare notevolmente la robustezza del codice [1]. Infatti, con i tipi è possibile dichiarare cosa esattamente una funzione deve ritornare e il tipo del parametro che si aspetta: in questo modo, già in fase di compilazione emergono eventuali errori frequenti durante la fase di scrittura. Inoltre, avere un linguaggio con la tipizzazione statica aiuta gli sviluppatori a comprendere velocemente il codice semplificando anche l'integrazione di librerie esterne.

Avere i tipi sulle variabili dà anche un livello di sicurezza molto superiore perché eventuali altri valori non sono accettati.

Quando si dichiara una variabile il tipo può essere assegnato automaticamente alla variabile in base al valore inserito (*types by inference*) oppure può essere dichiarato manualmente quando si vuole lavorare con tipi complessi, creati appositamente per le esigenze del progetto.

I tipi di base utilizzati da Typescript sono: *number*, *string* e *boolean*. Gli sviluppatori hanno poi la possibilità di creare ulteriori tipi tramite l'utilizzo di due sintassi: *Interface* e *Type*. Nell'esempio in figura 5 si vede come utilizzare entrambi le sintassi: per la maggior parte dei casi d'uso sono corretti entrambi. L'unica differenza tra i due è che utilizzando le interfacce è possibile dichiarare nuove interfacce estendendone altre già dichiarate tramite l'utilizzo di *extends*.

Figura 5 – Implementazione di un tipo e di un'interfaccia in Typescript.

Inoltre, Typescript, a differenza di Javascript, introduce gli strumenti necessari per poter utilizzare il *pattern* della programmazione orientati agli oggetti. In particolare, si utilizza il concetto di classe per creare tipi di oggetti che possiedono proprietà e metodi ereditabili. In più, questo permette di avere il codice molto più organizzato dal punto di vista dei dati e dà la possibilità di scrivere codice maggiormente robusto con una bassa probabilità di errore. Nell'immagine di seguito, viene mostrato come implementare una classe creando quindi un oggetto che implementa al suo interno alcune proprietà ed anche un metodo.

```typescript
class Person {
    name: string;
    surname: string;

    constructor(name: string, surname: string){
      this.name = name;
      this.surname = surname;
    }

    showNameSurname() {
        return this.name + ' ' + this.surname;
    }
}
var marioRossi = new Person('Mario', 'Rossi');

console.log(marioRossi.showNameSurname()); //Mario Rossi
```

Figura 6 - Esempio di creazione di una classe in Typescript.

Infine, il codice scritto in Typescript per poter essere usato nei *browser* o in ambienti che utilizzano sotto ambienti compatibili con Javascript, come NodeJS, deve essere compilato e trasformato in codice Javascript. Per questa sua piena compatibilità è preferito dagli sviluppatori: è possibile, infatti, sviluppare in modo sicuro e robusto sfruttando gli strumenti funzionanti con Javascript.

I vantaggi dell'utilizzo di Typescript sono molteplici, tuttavia, occorre considerare il fatto che scrivere del codice tipizzato richiede più tempo rispetto ad altri linguaggi con tipizzazione dinamica, poiché bisogna specificare i tipi. Per progetti individuali più piccoli potrebbe non valere la pena usarlo. Inoltre, ciascun progetto Typescript deve essere compilato effettuando una traduzione in Javascript. Questo, in base alla dimensione del progetto, potrebbe impattare sulla *performance*.

Oltre ai vantaggi elencati, si è deciso di utilizzare Typescript poiché durante la comunicazione tra i microservizi riesce a garantire, già nella fase di compilazione, che i dati che vengono inviati e/o ricevuti rispettino il tipo di dato che ci si aspetta. Questo fattore dà ulteriore robustezza all'intero sistema poiché in modo automatico viene eseguito il *type-checking* durante la lettura o la scrittura di un dato.

2.2.2 NestJs

NestJs è un *framework* che facilita la creazione di applicazioni *server-side* altamente efficienti e scalabili sfruttando le potenzialità di NodeJs. È scritto interamente in Typescript ma comunque dà la possibilità agli sviluppatori di utilizzare Javascript puro. Per fornire le funzionalità di un *server*, NestJs al suo interno utilizza robusti *framework* HTTP *Server* come Express. Sopra di essi NestJs fornisce un livello di astrazione superiore ma espone anche le loro API direttamente allo sviluppatore. Questo dà piena libertà agli sviluppatori di utilizzare moduli di terze parti per le piattaforme sottostanti.

Sebbene Javascript, grazie a NodeJs, negli ultimi anni sia diventato il punto di riferimento per la creazione di applicazioni *back-end* consentendo l'utilizzo di numerosissime librerie di terze parti, rimane aperto il problema principale mai risolto da NodeJs: avere un'architettura applicativa pronta all'uso per creare applicazioni altamente testabili, scalabili e di facile manutenzione [11].

In particolare, i componenti fondamentali dell'architettura di un progetto in NestJs sono i seguenti:

- *Controllers*: lo scopo principale dei *controller* è quello di elaborare le richieste provenienti dall'esterno per l'applicazione tramite un meccanismo di *routing*. Grazie all'uso del *decorator* "@Controller" è possibile associare una *route* ad una classe. Ciascun *controller* generalmente racchiude più *route*, e ciascuna di esse corrisponde ad una determinata funzionalità. Nell'esempio in figura 7 si vede come un *controller* viene dichiarato. La particolarità sta nel fatto che grazie al *decorator* "@Controller" è possibile raggruppare tutte le *route* che contengono nel *path* "/cats". In questo modo, all'interno della classe è possibile dichiarare i metodi corrispondenti alle richieste http che possono essere effettuate.

```
import { Controller, Get } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get()
  findAll(): string {
    return 'This action returns all cats';
  }
}
```

Figura 7 - Esempio di implementazione di un Controller.

- *Services*: i servizi in NestJs sono responsabili della memorizzazione del dato e del recupero del dato. Quindi, agiscono ad un livello inferiore rispetto ai *controller* e sono fatto proprio per essere usati dai *controller*. Essi, infatti, inglobano la logica applicativa del *software*. La caratteristica principale è che possono essere "iniettati" come una dipendenza nelle classi dove devono essere utilizzati. In questo modo, per esempio, i *controllers* possono esporre funzionalità implementate nei *providers*.
- *Modules*: i moduli sono delle classi utilizzate da NestJs per organizzare i componenti dell'applicazione. Allo *startup* del progetto, in automatico viene creato un *root module*. Secondo la logica di NestJs è fortemente consigliato creare un modulo per raggruppare un insieme di funzionalità strettamente correlate. Ciascuna classe possiede il *decorator* "@Module" che fornisce le informazioni per consentire a NestJs di organizzare le varie parti dell'applicazione.

I principali vantaggi dell'utilizzo di NestJs sono i seguenti:

- Architettura predefinita: come già anticipato all'inizio di questo paragrafo, NestJs fornisce una struttura del progetto predefinita che permette di concentrarsi direttamente sulla scrittura del codice velocizzando e semplificando il lavoro agli sviluppatori.
- Piena compatibilità con Typescript: l'uso di Typescript rende ancora di più sicuro il codice per i controlli che vengono effettuati sui tipi prima ancora di compilare il codice.
- Uso della *Dependency Injection*: si tratta di un *pattern* della programmazione orientata agli oggetti che permette di scrivere codice disaccoppiato e testabile. È una tecnica che permette ad un oggetto di delegare il compito di fornire le dipendenze necessarie ad un oggetto esterno. In NestJs, è possibile dichiarare un servizio *Injectable* così da poter essere iniettato in altri componenti. Così facendo ogni volta, il componente *Injectable* sarà responsabile di risolvere le dipendenze mantenendo quindi alto il livello di disaccoppiamento tra i vari moduli.

- CLI: NestJs fornisce anche una CLI, *Command Line Interface*, per la creazione dei componenti descritti. Anche questa è una caratteristica che avvantaggia gli sviluppatori durante il loro lavoro.

All'interno di questo progetto, questo *framework*, è stato scelto innanzitutto per la sua piena compatibilità con il linguaggio Typescript e anche perché con le *utility* messe a disposizione hanno permesso di implementare facilmente un *server web* in grado di ricevere delle richieste dagli altri microservizi. Oltre a ciò, NestJs mette a disposizione dei *decorator* per la creazione di *jobs* automatici: questo ha permesso di implementare le funzionalità automatiche definendo il *decorator* "@Cron()" prima dei metodi automatici.

2.2.3 NodeJs

NodeJs è un *runtime environment*, cioè un'ambiente di esecuzione *open-source* costruito sul motore JavaScript V8 di Google Chrome che permette di eseguire codice Javascript [12]. NodeJs nasce con l'obiettivo di usare JavaScript anche per scrivere codice da eseguire lato *server* perché prima della sua nascita veniva utilizzato soprattutto lato *client* con degli *script* elaborati da pagine HTML. Per questo motivo, infatti, oggi viene usato per creare applicazioni unificando l'intero sviluppo attorno ad un unico linguaggio di programmazione, cioè Javascript [13].

L'architettura di NodeJs è *Single Threaded Event Loop* e viene usata per gestire la concorrenza degli eventi, cioè le richieste eseguite dai vari *client*. In particolare, alloca un *event loop* che gestisce le varie richieste e se arriva una richiesta bloccante questa viene gestita da un *pool* di *thread*. In questo modo la gestione delle richieste è asincrona facilitando notevolmente la concorrenza perché riesce a gestire un elevato numero di richieste senza mai rallentare il processo di esecuzione.

Nodejs ha attorno a sé una grande comunità di sviluppatori di tutto il mondo che in maniera molto attiva realizza supporti e librerie che possono essere utilizzate per ampliare sempre di più i campi di applicazione. L'ecosistema che raccoglie tutte queste librerie si chiama *Node Package Manager* (npm): si tratta di un gestore di pacchetti che semplifica di gran lunga la vita al programmatore. Inoltre, si tratta di un *software* multipiattaforma, cioè, è disponibile su tutti i sistemi operativi.

Tuttavia, NodeJs presenta alcuni svantaggi: non è adatto per tutti i campi dell'informatica. Per esempio, per il *machine learning* è preferibile usare altri tipi di linguaggi. Inoltre, a causa della sua architettura quando si eseguono dei *task* molto complessi le *performance* calano drasticamente impattando sul funzionamento del programma. NodeJs inoltre è compatibile al 100% con il linguaggio Typescript utilizzato per questo progetto poiché si tratta di un *superset* di Javascript. Infatti, il codice viene prima compilato e tradotto in Javascript, quindi non c'è nessun problema di compatibilità.

Per questo progetto di tesi si è deciso di utilizzare NodeJs perché si tratta di una tecnologia molto moderna che permette di programmare in maniera molto innovativa grazie alle librerie sempre aggiornate dalla comunità di sviluppatori. Inoltre, le applicazioni in NodeJs sono molto performanti rispetto all'utilizzo di tecnologie con altri linguaggi di programmazione.

Ad oggi Node.js è sicuramente tra le tecnologie più sfruttate dagli sviluppatori. Secondo diversi *report* nel 2022 il 46% degli sviluppatori ha usato a livello professionale almeno una volta il *framework* NodeJs [14].

2.2.4 MongoDB

MongoDB è un *database open-source* non relazionale orientato ai documenti sviluppato in C++; quindi, i dati vengono memorizzati all'interno di documenti e non in righe di tabelle come avviene nei *database* relazionali. Inizialmente è stato sviluppato da una società di *software* nel 2007 come un componente di un prodotto di *platform as a service*. Nel 2009 l'azienda si è spostata verso una soluzione *open-source* dando vita a MongoDB. Da allora è adottato da numerosissime aziende, tra cui Ebay [15]. Ad oggi, è il quinto *database* più popolare al mondo e il primo tra quelli NoSQL [16].

Il motivo per cui nascono e si sviluppano i *database* non relazionali, detti anche NoSQL, è che negli ultimi anni stanno crescendo sempre di più i *BigData*. Avendo tanti dati da gestire provenienti da diverse parti del *web*, i classici *database* relazionali con schemi fissi diventano poco funzionali rispetto alla grande variabilità e diversità di dati che i sistemi devono elaborare. Infatti, per definizione, i *database* non relazionali scardinano lo schema fisso di tabelle e colonne creando collezioni di dati con uno schema variabile.

Per la facilità di sviluppo e le prestazioni che riescono a raggiungere si sono sviluppati molto velocemente.

Un singolo dato in MongoDB viene memorizzato sotto forma di documento, cioè una struttura dato composta da coppie campo-valore: si tratta di un formato molto simile al JSON, chiamato BSON (*Binary JavaScript Object Notation*, il che significa che i campi possono variare da un documento all'altro ed è possibile modificare nel tempo la struttura dei dati. Inoltre, questa struttura permette di associare ad un campo un documento, creando quindi documenti annidati. Oltre a questa caratteristica, MongoDB presenta altre importanti caratteristiche [17]:

- *Query* API: MongoDB offre numerosi operatori e funzioni di *query* per recuperare e manipolare i dati. Fornisce *query ad hoc* per effettuare aggregazioni, *join* e altre operazioni più complesse.
- Alta disponibilità: per garantire un'elevata disponibilità dei dati anche in presenza di guasti, esso supporta la replicazione dei dati; quindi, i dati vengono copiati su più macchine sparse per garantire la ridondanza e la tolleranza ai guasti. In caso di guasto, MongoDB può automaticamente recuperare una copia dei dati e fornirla a chi l'ha richiesta per garantire la continuità del servizio.
- Scalabilità orizzontale: MongoDB è progettato per la scalabilità orizzontale; quindi, è in grado di gestire grandi volumi di dati distribuendo i dati su più *server*. Per questo motivo, infatti, si presta bene per gestire elevati carichi di lavoro garantendo elevate prestazioni.
- Alte prestazioni: grazie alla scalabilità orizzontale, descritta al punto precedente, unita all'indicizzazione efficiente e ad ottimizzazioni specifiche per le operazioni di scrittura, la velocità di esecuzione delle *query* è molto elevata e permette di avere un alto livello di affidabilità del sistema.

Tuttavia, per certi aspetti, ha comunque degli svantaggi. In particolare, MongoDB non supporta le transazioni come i classici DBMS, *database management system*: questo, quindi, limita le funzionalità del *software* nel caso in cui si vuole implementare un sistema di transazioni fatto da operazioni atomiche. Inoltre, non è supportata l'operazione di JOIN, non dando la possibilità quindi di creare delle *query* che interrogano più *collection*.

Infine, ha un *data size* maggiore perché occorre memorizzare tutti i campi e tutti i valori per ciascun documento.

Per garantire la persistenza nel *software* sviluppato per questo progetto è stato scelto proprio per queste caratteristiche appena elencate. Ma, oltre a ciò, il motivo per cui si è deciso di utilizzarlo riguarda la sua facilità di uso e compatibilità con il *framework* NestJs, poiché quest'ultimo mette a disposizione un *tool*, chiamato *Mongoose*, che facilita la gestione del *database* che deve interagire con i servizi implementati.

2.2.5 Docker e Kubernetes

Docker [18] è un *software opensource*, sviluppato in Go, che permette di eseguire applicazioni in ambienti virtualizzati tramite l'utilizzo di *Linux Containers* (LXC). Ciascuno simula il comportamento dell'applicazione su un ambiente isolato e indipendente dalla macchina in cui si esegue. Questo facilita il processo di sviluppo perché permette di analizzare il comportamento dell'applicazione cambiando le dipendenze. I principali componenti di Docker sono i seguenti:

- **Docker Engine**: rappresenta la parte centrale del *software* e offre tutti i servizi necessari per creare e mandare in esecuzione i *container*. In particolare, mette a disposizione degli sviluppatori un *set* di API per gestire i *container* e le risorse associate.
- **Docker Client**: è un'applicazione fornita da Docker che consente di interagire in modo semplice con le API messe a disposizione per la creazione e la gestione dei *container*. Il Docker Client fornisce un'interfaccia da riga di comando (CLI) intuitiva che consente agli utenti di eseguire comandi per gestire i *container* Docker. Attraverso il Docker Client, è possibile avviare, fermare, riavviare, eliminare e controllare lo stato dei *container*. Inoltre, il Docker Client permette anche di gestire le immagini Docker, eseguire il *build* di nuove immagini, caricarle e scaricarle dai registri Docker. In sostanza, il Docker Client è uno strumento essenziale per l'interazione con il sistema Docker.
- **Docker Images**: si tratta di file che contengono le informazioni necessarie a creare un'istanza di un'applicazione all'interno di un *container*. Esse infatti definiscono la posizione del codice sorgente e tutte le dipendenze necessarie.

Inoltre, sono memorizzabili all'interno di registri pubblici o privati, quindi sono portabili.

- **Docker container**: un *container*, come già detto, è un'istanza di un'applicazione creato a partire da un'immagine. Essi vengono creati tramite l'operazione di *build* che viene eseguita su un'immagine. Come risultato si ha un pacchetto che contiene tutto il necessario per avviare l'applicazione in qualsiasi ambiente.

Negli ultimi anni Docker è diventato molto popolare e preferito dalle aziende per i vantaggi che i *container* Linux offrono. I principali vantaggi sono i seguenti [19]:

- **Velocità**: il tempo richiesto per la costruzione di un *container* è molto basso. Questo consente di accelerare le fasi di creazione di un *software*, a partire dallo sviluppo fino alla consegna finale.
- **Portabilità**: i *container* permettono di eseguire facilmente la stessa applicazione in ambienti diversi indipendentemente dal sistema operativo e dall'*hardware* della macchina.
- **Scalabilità**: i singoli *container* possono essere rilasciati su qualunque piattaforma. In questo modo, è possibile scegliere in base alla necessità le risorse per ciascun *container*.
- **Consegna rapida**: l'operatività tra gli amministratori di sistema e i programmatori è molto più veloce perché i *container* sono standardizzati. Quindi, dopo aver sviluppato e testato l'applicazione, occorre solo rilasciarla in produzione. Questo accelera di gran lunga l'intero processo di sviluppo.
- **Densità**: dato che si possono eseguire più *container* sullo stesso *host*, le prestazioni sono nettamente migliori rispetto all'utilizzo di eventuali *virtual machine*.

Per lo sviluppo di un applicativo con un'architettura formata da microservizi, Docker è lo strumento ideale per gestire un *software* che è formato da tanti servizi perché permette di sviluppare, modificare ed eseguire il *deploy* in modo indipendente. Per questo motivo, si è utilizzato per lo sviluppo di ciascun microservizio di questo progetto di tesi. In particolare, per ciascun microservizio è stata definito un Dockerfile, utilizzato dalla *pipeline* della *Continuous Integration* e *Continuos Delivery* per creare il *container* per effettuare il rilascio in produzione e successivamente il *deployment* finale.

I Dockerfile sono dei documenti di testo speciali impiegati per definire e costruire le immagini Docker. Un Dockerfile contiene una serie di istruzioni che descrivono passo dopo passo come configurare l'ambiente all'interno di un *container* Docker. Queste istruzioni comprendono la selezione di un'immagine di base, l'aggiunta di dipendenze, la configurazione delle variabili d'ambiente, l'esecuzione di comandi e la specifica di come avviare l'applicazione all'interno del *container*. Una volta creato un Dockerfile, è possibile utilizzarlo come *input* per il comando "docker build", il quale genererà un'immagine Docker pronta per essere eseguita come *container* isolato. I Dockerfile rappresentano uno strumento essenziale per creare immagini Docker riproducibili, scalabili e portabili.

Per l'orchestrazione dei vari *container* è stato utilizzato Kubernetes, abbreviato K8s, è una piattaforma *opensource* utilizzata per la gestione e l'orchestrazione di *workload*. È stato sviluppato inizialmente da Google e rilasciato come progetto *opensource* nel 2014 ed è una tecnologia complementare a Docker perché esegue e gestisce l'esecuzione di *container*. L'utilizzo di quest'ultimi per il *deployment* di applicazioni è una soluzione molto efficiente: le applicazioni virtualizzate tramite i *container* non condividono lo stesso sistema operativo a differenza degli approcci tradizionali con le *Virtual Machine* oppure con l'utilizzo di *server* fisici, piuttosto ogni servizio usa un sistema operativo minimale a lui dedicato [20]. Per lo sviluppo di applicazioni con i microservizi ha un ruolo centrale perché si occupa di orchestrare i vari servizi che sono in esecuzione. Viene data la possibilità di combinare liberamente i microservizi e di avere alta scalabilità. Questo consente di avere l'efficienza massima dal *software* implementato con l'architettura a microservizi. Utilizzare Kubernetes consente di usufruire dei seguenti vantaggi [21]:

- Operazioni automatizzate: fornisce dei comandi per automatizzare operazioni da eseguire quotidianamente.
- Astrazione dell'infrastruttura: dopo aver installato K8S, la gestione del calcolo, il *networking* e l'archiviazione dei *workload* vengono fatte in maniera automatica consentendo agli sviluppatori di non preoccuparsi dell'infrastruttura sottostante.
- Monitoraggio dell'integrità dei servizi: Kubernetes esegue in modo automatico e ripetuto controlli sui *container*. In particolare, esegue controlli di integrità sui

servizi, e se c'è qualche *container* con errori lo riavvia rendendo i servizi disponibili agli utenti il prima possibile.

Per quanto riguarda il progetto di tesi, si è scelto di utilizzare Kubernetes come orchestratore dei microservizi poiché è parte integrante dell'ecosistema di Google Cloud.

3. Analisi

In questo capitolo verrà fatta un'analisi del *software* dapprima descrivendo le funzionalità del *chatbot*. Successivamente, dopo aver analizzato i servizi esterni coinvolti nel funzionamento dell'intero sistema, verranno descritti i limiti dell'utilizzo dell'architettura monolitica e gli obiettivi posti per questo progetto di tesi.

3.1 Funzionalità

Il *software* implementato per questo progetto di tesi si colloca direttamente nel contesto aziendale assumendo il ruolo di assistente a fianco del programmatore durante il suo lavoro quotidiano. Ciascun dipendente all'interno dell'azienda, in base al proprio ruolo, ha la possibilità di accedere e lavorare con i progetti aziendali memorizzati su Gitlab.

È opportuno che ciascun progetto, in fase di creazione, abbia una determinata configurazione scelta secondo gli *standard* aziendali. Uno dei compiti del *software* è di controllare quotidianamente, nelle ore non lavorative, le configurazioni di tutti i progetti e aggiornarle nel caso in cui queste non dovessero essere in linea con gli *standard* decisi: questo viene eseguito in modo automatico affidando, quindi, la responsabilità di configurare i progetti all'assistente virtuale. In ogni caso, il programmatore dopo aver creato un nuovo progetto può configurarlo manualmente senza aspettare che l'applicativo lo faccia autonomamente. Questo può essere fatto lanciando dei comandi appositi che il *software* mette a disposizione tramite Slack, che eseguono, le stesse operazioni eseguite durante le ore non lavorative. Infine, il *software* ha anche il ruolo di *reminder*: si occupa,

infatti, di ricordare ai singoli dipendenti tramite l'invio di notifiche le scadenze per la gestione del *timesheet*.

Poiché esegue operazioni automatiche e dà la possibilità all'utente di eseguire comandi mandando dei semplici messaggi in *chat* tramite Slack, integrandosi quindi come un vero utente, può essere definito *bot*, o *chatbot*. Le funzionalità del *bot* si possono dividere in due macroaree:

- Servizi automatici: racchiudono le operazioni che in modo automatico e indipendente vengono eseguiti in *background* nelle ore non lavorative e non richiedono nessuna interazione da parte dell'utente.
- Comandi: si riferiscono alle istruzioni che l'utente può fornire tramite la piattaforma Slack per lanciare determinate operazioni descritte successivamente.

3.1.1 Servizi automatici

I servizi automatici si possono classificare a sua volta in:

- Notifiche, cioè *reminder* che vengono inviati ai dipendenti in determinati canali dell'area di lavoro aziendale di Slack.
- Configurazioni, cioè dei servizi che in modo automatico si occupano di eseguire le configurazioni descritte nel paragrafo delle funzionalità.

Notifiche

Ciascun dipendente dell'azienda per ogni mese lavorativo deve compilare e inviare il *timesheet* con le ore di lavoro svolte. È compito del *bot* inviare delle notifiche agli utenti per ricordarlo. Per questo vengono inviate tre notifiche descritte nella tabella seguente.

Reminder timesheet	descrizione notifica
Compilazione settimanale	Notifica inviata ogni venerdì alle 17:00 per ricordare di compilare il *timesheet* con le ore svolte durante la settimana.
Invio	Notifica inviata nell'ultimo giorno lavorativo del mese alle ore 17:00 e ricorda di inviare il *timesheet* compilato nelle settimane del mese trascorso.
Approvazione	Notifica inviata il primo giorno lavorativo alle 11:00 per ricordare di approvare il *timesheet* con le ore svolte il mese precedente.

Configurazioni

Ciascun programmatore per svolgere il suo lavoro ha a disposizione dei progetti su Gitlab che contengono tutto il codice creato. Ogni progetto segue un flusso di lavoro ben preciso e per questo occorre configurare le impostazioni di ogni progetto. In particolare, il *bot* esegue le seguenti operazioni:

- Configurazione "merge method": il *merge* è un'operazione che permette di integrare le modifiche di due linee di sviluppo quando si lavora con *repository* Git. Gitlab, sfruttando la potenzialità di Git, mette a disposizione tre modalità per decidere come integrare le nuove modifiche nel *branch* esistente:
    - "Merge Commit": selezionando questa modalità ogni volta che viene eseguito il *merge* viene creato e aggiunto un nuovo *commit* nel ramo di sviluppo principale.
    - "Merge commit with semi linear history": anche con questa modalità viene creato un *merge commit* per ogni operazione di *merge*. Tuttavia, è possibile integrare le modifiche delle due linee di sviluppo solo nel caso in cui non

ci siano conflitti tra i due rami, cioè quando è possibile eseguire il *fast forward merge*.

- "Fast forward merge": questa tecnica consiste nell'unire due rami di sviluppo semplicemente sincronizzando le cronologie dei *commit* dei due *branch* senza creare un nuovo *merge commit*. In questo modo, viene mantenuta pulita la *history* dai *commit* di *merge*. Questo tipo di *merge* è possibile eseguirlo solamente quando le due linee di sviluppo non sono divergenti. Eventualmente si è costretti ad eseguire il *rebase* rispetto al *branch* con il quale si vuole effettuare il *merge*.

Poiché i *team* di sviluppo hanno l'obiettivo di mantenere la cronologia dei *commit* pulita e lineare, la metodologia "Fast-Forward merge" [22] è stata scelta per eseguire il *merge*.

- Configurazione "squash option": per ciascun progetto, Gitlab dà la possibilità di configurare quale deve essere il metodo di *default* per l'esecuzione dello *squash* dei *commits* quando viene eseguita l'unione di due flussi di lavoro, cioè, quando viene eseguito il *merge*. Lo *squash* letteralmente indica lo "schiacciamento" dei *commit*, cioè l'unione di più *commit* in uno singolo. Questa operazione viene fatta per evitare di ereditare l'intera cronologia dei *commit*. Il compito del *software* è quello di configurare la possibilità di eseguire lo *squash* dei *commit* rendendolo obbligatorio al programmatore [22].
- Configurazione "protected branches": Gitlab inoltre dà la possibilità di applicare restrizioni ai *branch*. In questo caso vengono marcati come *protected* per evitare che chiunque possa eseguire operazioni sul *branch* protetto ma solamente chi è stato autorizzato.

Branch	Allowed to merge	Allowed to push	Allowed to force push
main	Mantainers	No one	✗
.-stable	Mantainers	No one	✗
environment-branch	Mantainers	No one	✗

Figura 8 - Tabella riassuntiva delle restrizioni applicate in base al tipo di branch.

Nella figura 8 è riportata la tabella nella quale sono descritti i *branch* da marcare come *protected* con le relative restrizioni. In particolare, le restrizioni applicate riguardano le seguenti operazioni:

- "Allowed to merge": quando ad uno sviluppatore viene dato il permesso di eseguire il *merge*, vuol dire che può unire le proprie modifiche con un'altra linea di sviluppo, tipicamente con il *main*. Per decisioni aziendali, questo permesso viene dato solo ai *maintainers* del progetto.
- "Allowed to push": questo permesso dà la possibilità di decidere chi può eseguire l'operazione di *push* su un *repository* remoto; così l'amministratore di sistema può decidere che ha il permesso di inviare le proprie modifiche nel *branch*. Nel caso di un *repository* aziendale, quindi condiviso, non viene assegnato nessun permesso particolare.
- "Allowed to force push": alcune operazioni in Git richiedono di forzare un *push* per consentite la sovrascrittura della storia del *repository* remoto con una versione. Occorre considerare il fatto che si possono perdere eventuali dati in modo permanente, per questo motivo i *protected branches* non permettono il *force push.*

Gitlab dà, inoltre, la possibilità di utilizzare le *wildcards*, cioè utilizzare l'asterisco per selezionare tutti quei *branch* che hanno quel formato. Per questo motivo, infatti, tutti i *branch* che rispettano il formato della seconda riga avranno applicate quelle restrizioni.

- Configurazione "protected tags": così come i *branch*, anche alcuni *tag* (si tratta di etichette associate ai *commit* usate principalmente per versionare il progetto a piacimento) vengono marcati come *protected*, ad esempio per impedire rilasci incontrollati. In questo caso, come si vede nella tabella rappresentata in figura 9, tutti i tipi di *tag* possono essere creati dai *maintainers* del progetto, e chi ha il ruolo di *developer* può solo creare *tag* che rispettano la *wildcard* "*-dev*".

Protected tags	Allowed to create
-dev	Developers+Mantainers
-rc	Mantainers
*	Mantainers

Figura 9 – Tabella riassuntiva dei permessi assegnati al tag dei progetti aziendali caricati sui repository Gitlab.

- Cancellazione immagini "container registry": quest'ultimo è un *repository* collegato al progetto che contiene le immagini Docker generate per ciascuna versione del *software*. Questo registro permette di archiviare in modo sicuro le immagini e permette l'accesso da diverse piattaforme. Nel contesto aziendale, per ogni versione del progetto rilasciata viene creata un'immagine Docker e viene memorizzata in questo *storage*. Per questo vengono eseguite periodiche operazioni di pulizia, lasciando solo le ultime cinque per ciascun ambiente di lavoro.
- Cancellazione *cache* Google Cloud Console: le *pipeline* di *Continuous Integration and Continuous Delivery* (CI/CD) utilizzano molta *cache* e conservano i dati in un *bucket* su Google Cloud Platform. Per svuotarla, ogni prima e terza domenica del mese, un servizio automatico pulisce tutta la memoria del *bucket*.

3.1.2 Comandi

Per quanto riguarda i comandi messi a disposizione dal *bot* l'utente può:

- Configurare il "gitlab access token": ciascun utente che vuole utilizzare i servizi che interagiscono con Gitlab deve prima configurare il proprio *token*; quindi, dopo averlo creato, lo comunica al *bot* così da poter essere autorizzato ad eseguire i comandi successivi. Questa è un'operazione obbligatoria perché il *bot* deve conoscere chi vuole eseguire la configurazione, soprattutto per verificare che l'utente abbia permessi sufficienti.

- Configurare il progetto: come già detto in precedenza, l'utente quando crea un nuovo progetto, per configurare il progetto può decidere di attendere l'esecuzione automatica o procedere manualmente. In particolare, lanciando il comando di configurazione si marcano come *protected* determinati *branch* e *tag*. Inoltre, viene selezionata la modalità preferita per eseguire il *merge* e come deve essere settata l'impostazione dello *squash* dei *commit*.
- Configurare la variabile del progetto "bot ignore settings": in azienda, può nascere la necessità di avere un progetto con una configurazione diversa. Per questo motivo, si sfrutta la possibilità che offre Gitlab di creare una variabile per memorizzare un booleano. In base al valore, ogni volta che viene avviato il servizio responsabile della configurazione dei progetti, si effettua un controllo della variabile booleana per controllare se quel progetto deve essere configurato o meno.

3.2 Servizi esterni

In questo paragrafo sono descritti i servizi esterni con i quali il *software* ha la necessità di interagire per poter implementare tutte le funzionalità richieste. Per questo motivo, infatti, vengono utilizzate le APIs, *Application Programming Interface*, interfacce esposte dai servizi esterni che permettono di comunicare con il *software*.

3.2.1 Slack

Slack è un'applicazione di messaggistica *cloud based* utilizzata dalle aziende e ha come obiettivo quello di aumentare l'efficienza e la produttività del *team working*. Infatti, permette la condivisione di messaggi in modo molto veloce e, grazie alla possibilità di creare canali, è possibile discutere su determinate aree tematiche e condividere contenuti riguardanti progetti specifici. Slack, inoltre, permette di organizzare le conversazioni in *thread* consentendo quindi di rispondere ai messaggi inviati senza riempire di messaggi la *chat*. Molte aziende scelgono di usare Slack anche per la sua facile integrazione con strumenti di terze parti.

Per esempio, nella *workspace* aziendale è comune l'uso di Google Calendar, un'applicazione Slack offerta da Google che permette di integrare il *calendar* di Google e quindi di ricevere *reminder* relativi agli eventi caricati nel calendario personale.

Per il progetto di tesi, viene usata questa possibilità di integrare nuove applicazioni. Essa mette a disposizione gratuitamente agli sviluppatori API e strumenti per permettere l'accesso alla scrittura e alla lettura dei messaggi nelle conversazioni degli utenti fisici [23], ma dà anche la possibilità di creare dei comandi per eseguire diverse azioni e di rispondere a eventi. Grazie a queste funzionalità è possibile quindi creare *bot* per la piattaforma.

3.2.2 Gitlab

Gitlab è una piattaforma *web opensource* utilizzata per la creazione e gestione di progetti tramite l'utilizzo di *version control system (VSC)* Git [22] e i suoi *repository*; essa permette di seguire l'intero processo di sviluppo di un prodotto dall'analisi al rilascio consentendo a più persone di lavorare contemporaneamente sugli stessi progetti.

Gitlab mette a disposizione degli sviluppatori le API che consentono di interagire con il *back-end* della piattaforma dando quindi la possibilità di automatizzare, integrare ed estendere le funzionalità di base offerte da Gitlab [24]. In particolare, per il progetto di tesi, sono state utilizzate le API che permettono di gestire i progetti sia per la creazione sia per la configurazione del progetto stesso.

3.2.3 Google Cloud Platform

La Google Cloud Platform (GCP) è una piattaforma di Google che offre servizi di *cloud computing*. Lo scopo è quello di offrire servizi e risorse utilizzabili sfruttando *internet* evitando quindi l'utilizzo di sistemi fisici che richiedono sia spazio adeguato sia risorse economiche per poterlo gestire. Google Cloud, sfruttando una *suite* di servizi *cloud*, dà la possibilità agli sviluppatori di creare, testare e rilasciare applicazioni utilizzando infrastrutture altamente scalabili e soprattutto robuste.

La Google Cloud Platform, oltre ad essere impiegata nella fase di rilascio, è utilizzata a sua volta dal *software* per interagire con diversi prodotti, in particolare con un *bucket*, cioè un sistema di *mass storage* in grado di memorizzare diversi tipi di dati non strutturati.

3.3 Analisi architettura monolitica

In questo paragrafo viene analizzato il *software* da cui parte il processo di migrazione verso la nuova architettura. Verrà presentata la sua struttura a livelli descrivendo il ruolo di ciascun modulo analizzando i limiti e gli obiettivi di questa soluzione.

3.3.1 Struttura

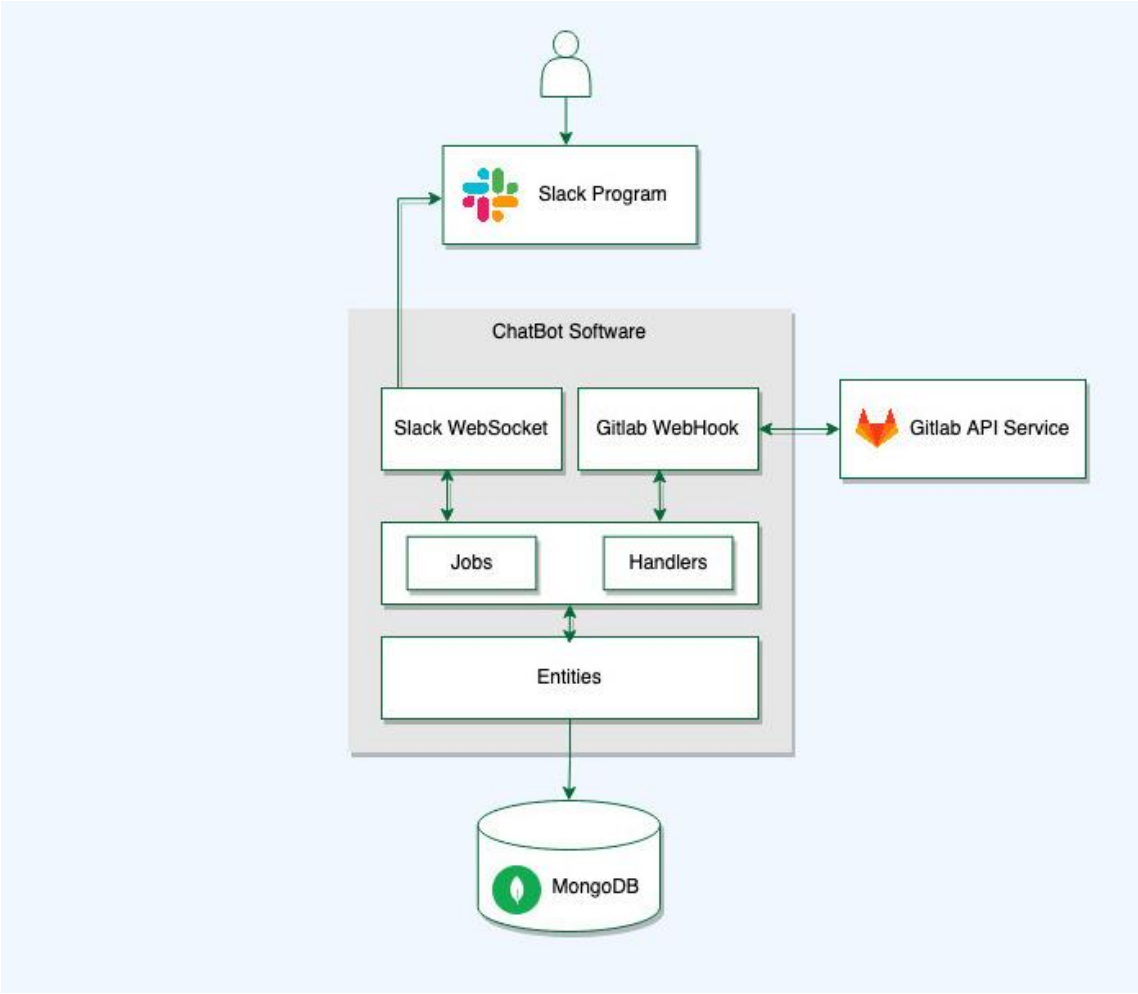


Figura 10 - Struttura del software monolitico analizzato.

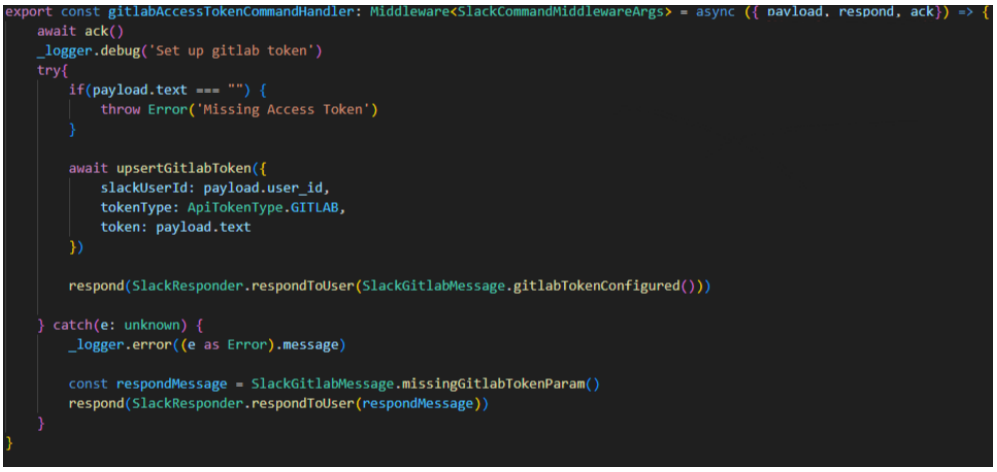
La figura 10 mostra gli elementi che fanno parte dell'architettura del *software* monolite. Il blocco chiamato "ChatBot Software" racchiude tutti i moduli che costituiscono il monolite. Questi sono raggruppabili nei tre livelli tipici dell'architettura stratificata:

- *Presentation Layer*: è il livello più alto che permette all'utente di potersi interfacciare con la logica del *software*. Fa parte di questo livello:
    - "Slack Program": inserita a questo livello perché fa riferimento alla Slack App installata nell'area di lavoro aziendale nella quale sono presenti le informazioni relative al *bot* e i comandi che sono messi a disposizione dal *software*. È l'unico punto in cui l'utente può interagire con l'applicativo.
    - "Slack Web Socket": permette di creare una comunicazione bidirezionale tra la macchina che ha in esecuzione il *software* e l'applicazione installata sull'area di lavoro di Slack utilizzando il *Web Socket Protocol* [25]. Dalla *dashboard* di creazione di applicazioni Slack, è possibile attivare la modalità "Socket Mode". Dopo averla attivata, è possibile configurare la propria applicazione con i dettagli del *web socket* così da poter creare la connessione.
    - Gitlab Web Hook: sono delle *callback* HTTP*, HyperText Transfer Protocol*, personalizzate e sono invocate quando si verificano alcuni eventi. Nel *software* analizzato sono stati utilizzati per catturare gli eventi che si verificano nella *pipeline* di sviluppo ed eseguire operazioni al verificarsi dell'evento. In particolare, sono state utilizzate qui per rispondere ad eventi di Gitlab per inviare notifiche su Slack [26].
- *Business Logic*: come si vede nella figura 10 questo livello racchiude due importanti moduli, *Jobs* e *Handlers*, che implementano la logica operativa:
    - *Jobs*: rappresentano la vera parte centrale del *software*. Infatti, contengono tutte le operazioni che devono essere svolte dal *bot*. All'interno di questo modulo sono presenti i file con i metodi che hanno lo scopo di salvare localmente le informazioni dei progetti memorizzati all'interno dei *repository* Gitlab e configurarli secondo i dettagli presenti nel paragrafo dove vengono descritte le funzionalità. L'implementazione dei *jobs*, cioè dei metodi che vengono eseguiti in modo automatico, è possibile grazie all'utilizzo della libreria "node-schedule" che dà la possibilità di eseguire

delle funzioni fornendo il momento esatto di esecuzione. Di seguito, è presente un esempio d'uso della libreria, che si occupa di lanciare il metodo *reminder*, sfruttando i dettagli di esecuzioni definiti nella *reminder rule*:

```
nodeSchedule.scheduleJob(reminderRule,reminder);
```

- *Handlers*: in questo modulo si trovano i file che hanno lo scopo di gestire la comunicazione tra il *software* e Slack; quindi, si tratta di metodi astratti che fungono da interfaccia verso la vera *business logic*. Infatti, in questo modulo sono presenti le funzioni collegate ai comandi presenti su Slack. Nella figura 11 si vede un esempio di *handler* che viene eseguito ogni volta che l'utente vuole configurare il proprio *gitlab access token*: viene lanciato un metodo della *business logic* che esegue una *upsert dell'access token*, cioè se presente lo aggiorna altrimenti lo aggiunge.



```
export const gitlabAccessTokenCommandHandler: Middleware<SlackCommandMiddlewareArgs> = async ({ payload, respond, ack}) => {
    await ack()
    _logger.debug('Set up gitlab token')
    try{
        if(payload.text === "") {
            throw Error('Missing Access Token')
        }

        await upsertGitlabToken({
            slackUserId: payload.user_id,
            tokenType: ApiTokenType.GITLAB,
            token: payload.text
        })

        respond(SlackResponder.respondToUser(SlackGitlabMessage.gitlabTokenConfigured()))

    } catch(e: unknown) {
        _logger.error((e as Error).message)

        const respondMessage = SlackGitlabMessage.missingGitlabTokenParam()
        respond(SlackResponder.respondToUser(respondMessage))
    }
}
```

Figura 11 - Metodo che esegue l'upsert del token Gitlab relativo ad un utente.

- *Data Access Layer*: in questo livello viene gestita la comunicazione con il *database*. In particolare, vengono definite le Entities, cioè i tipi di dato che devono essere inviati o letti dal *database*. L'applicativo deve formattare il dato in uscita correttamente e deve anche essere in grado di leggerlo correttamente dal *database* senza perdere nessuna informazione. In particolare, le entità utilizzate sono, "Group" e "Project", le quali sono state memorizzate nel *database* per svolgere le

operazioni all'interno della *business logic*. Esse fanno riferimenti ai gruppi contenenti i progetti che devono essere configurati.

Per garantire la persistenza dei dati è stato utilizzato una singola istanza di MongoDB, database non relazionale descritto al paragrafo 2.2.4.. All'avvio del progetto, viene eseguita la connessione al *database* fornendo lo *username* e la *password* per poter essere abilitati ad eseguire le operazioni su di esso.

3.3.2 Limiti

Generalmente un *software* può avere diversi tipi di limiti che influenzano le sue funzionalità e le sue prestazioni. Nel contesto del *software* monolite, analizzato per questo progetto di tesi, è emerso che, come tutti i *software* monolitici, anche questo presenta alcuni limiti dovuti principalmente al tipo di architettura utilizzato.

In particolare, dall'analisi effettuata si nota che la *business logic* del progetto iniziale è già logicamente divisa in due: ci sono infatti, allo stesso livello, servizi che implementano gli *Handlers* di Slack e i *Jobs* che comunicano con il *server* Gitlab per la configurazione dei progetti. All'interno di questo livello ci sono quindi funzioni che appartengono a spazi diversi ma altamente collegate tra di loro. Avere questo tipo di connessione a lungo andare aumenta la complessità del *software* perché diminuisce il disaccoppiamento tra le singole parti; infatti, se si apporta una modifica ad una piccola parte, occorre anche riadattare il resto del sistema per garantire il corretto funzionamento.

Inoltre, per l'intero progetto è stato utilizzato il *framework* NodeJS e Typescript come linguaggio di programmazione obbligando gli sviluppatori ad utilizzare in futuro queste due tecnologie. Questo aspetto, di fronte al grande numero crescente di tecnologie, risulta essere negativo poiché non è possibile scegliere una nuova tecnologia più performante e adatta per una nuova funzionalità che si presenta.

Inoltre, il progetto che contiene l'intero codice sorgente del *software* monolite contiene al suo interno tante cartelle con dentro a sua volta tanti file. Questo a lungo andare, man mano il numero di file cresce, può rendere il *software* difficile da gestire ma soprattutto complica il lavoro agli sviluppatori che non conoscono il *software*. A sua volta, questa

complessità si riflette nelle performance del *software*. Continuare a lavorare con questo codice monoblocco rallenta l'esecuzione del servizio.

3.3.3 Obiettivi

I limiti introdotti da quest'architettura nel contesto del *software* analizzato hanno fatto nascere l'esigenza di modernizzare l'intero progetto del *chatbot* utilizzando un'architettura diversa con la finalità di rendere l'applicativo più efficiente, stabile e sicuro. In particolare, gli obiettivi che si sono posti per la migrazione di questo *software* sono i seguenti:

- Usare uno *stack* tecnologico non fisso: l'obiettivo che ci si è posti è stato quello di avere la possibilità di utilizzare *framework* e/o linguaggi che si possono adattare meglio alle funzionalità che al momento sono implementate ma soprattutto per quei requisiti da implementare in un secondo momento.
- Separazione dei domini: a fronte dei limiti riscontrati ci si è posti l'obiettivo di evidenziare i limiti del dominio effettuando una marcata divisione. Questo per permettere di avere una piena consapevolezza dei limiti funzionali di ogni servizio e dell'intero sistema da migrare facilitando quindi la gestione delle dipendenze e le interazioni dei vari servizi/domini.
- Sviluppo microservizi piccoli: tra i limiti evidenziati nel *software* monolite emerge la presenza di numerosi file contenenti tante righe di codice. Per questo motivo, infatti, si è scelto di sviluppare microservizi piccoli per avere codice più mantenibile e soprattutto più comprensibile ad eventuali nuovi sviluppatori che si ritrovano a lavorare sul progetto di questa tesi.
- Velocizzare processo di sviluppo: per rendere il *software* moderno, è stato necessario progettare la migrazione per avere un processo di sviluppo veloce. Migrando il *software* da singolo monolite a più servizi, il processo di sviluppo non deve più seguire l'intero *software* ma solamente il singolo servizio su cui si lavora dando quindi la possibilità di parallelizzare il processo.
- *Software* resiliente: più un *software* è progettato per restare attivo anche in presenza di guasti più può essere definito resiliente. Per questo motivo, uno degli obiettivi di questa tesi è stato quello di isolare i servizi. Quindi, non solo

scomporre il *software* in microservizi, ma progettare ciascun microservizio in modo tale da essere autonomo e indipendente dagli altri servizi del sistema, sia interni che esterni.

4. Soluzione sviluppata

4.1 Analisi nuova architettura

Il primo passo per lo sviluppo della nuova architettura consiste nella scomposizione dei requisiti in più servizi ciascuno in grado di eseguire una parte delle funzionalità. In questo paragrafo, quindi, verranno presentati prima i metodi principali per decomporre un sistema e quale scelta è stata fatta per garantire la persistenza dei dati. Successivamente verranno descritte le strutture dei microservizi implementati e come questi interagiscono tra di loro.

4.1.1 Decomposizione in Microservizi

La decomposizione in microservizi è un processo finalizzato alla suddivisione del monolite con l'obiettivo di ottenere servizi coesi e disaccoppiati tra di loro. I metodi più usati sono i seguenti:

- **Decompose by subdomains**: consiste nel suddividere il dominio di un *software* ottenuto tramite il *domain-driven design* (DDD). Questo approccio aiuta a scomporre un dominio in gruppi indipendenti, detti sottodomini, ciascuno con le proprie responsabilità e *business rules*. Ad ogni sottodominio si associa un microservizio, ottenendo maggiore stabilità e servizi ben scollegati tra di loro. Tuttavia, un eventuale cambio di dominio richiederebbe drastiche modifiche nel codice.
- **Decompose by business capability**: la decomposizione in microservizi avviene analizzando la struttura e i processi dell'organizzazione. Si delineano quindi quali sono le *business capabilities* e per ciascuna viene creato un microservizio. I

risultati sono molto simili a quelli ottenuti con la decomposizione *by subdomains* ma c'è il rischio che un microservizio possa inglobare molte entità strettamente correlate tra di loro.

- **Self-contained:** per ogni funzionalità viene creato un microservizio; si tratta quindi di un approccio più schematico e meno dipendente dalle scelte aziendali. I microservizi eseguono le operazioni in modalità completamente asincrona. I tempi di risposta sono molto più bassi rispetto ai due approcci precedenti e la probabilità che il sistema vada in blocco diminuisce. Tuttavia, il costo è elevato e aumenta anche la complessità del *software*: occorre gestire la comunicazione tra i servizi in modo asincrono garantendo la consistenza dei dati.

Vista l'architettura del *software* monolitico nel paragrafo precedente, si è deciso di scomporre il monolite utilizzando lo schema *by subdomains*, perché il dominio mostra già come sia naturale la divisione in più moduli, Slack e Gitlab. Ciò significa che per molte parti non è necessario scrivere il codice da zero poiché occorre solo riadattare il codice esistente.

Per il *Data Access layer* occorre fare un'analisi diversa: il suo scopo è quello di interfacciarsi con il *database* che contiene informazioni relative ai progetti e ai gruppi di progetti di Gitlab. Oltre a queste informazioni tiene traccia delle utenze che tramite Slack vogliono interagire con Gitlab. Si pone quindi tra i due domini servendoli entrambi. Secondo il principio della decomposizione, tra i due servizi è possibile identificare un terzo sottodominio. Per questo, si è creato un terzo microservizio per gestire le operazioni per la memorizzazione e l'identificazione degli utenti.

4.1.2 Data management

La gestione dei dati in un'architettura a microservizi prevede la creazione di un *database* per ciascun servizio per rendere sempre più indipendente il singolo servizio dal resto dell'applicazione. Tuttavia, avere più *database* causa un aumento della complessità e del costo rispetto ad averne uno unico condiviso tra i vari servizi. Per esempio, quando ci sono delle transazioni che interessano più servizi non è più possibile usare una semplice transazione ACID (Atomicità, Coerenza, Isolamento e Durabilità). Per questo si utilizza

il SAGA *pattern*, cioè si implementa un sistema di transazioni locali e ogni volta che ognuna va a buon fine invia un messaggio che fa partire la successiva. Questo però richiede l'implementazione di operazioni di compensazione: se la singola transazione non va a buon fine si esegue un'azione che ha lo scopo di annullare il lavoro eseguito dall'operazione principale [27].

Inoltre, in un sistema distribuito le operazioni sui dati possono essere fatte da più servizi anche in parallelo. Per mantenere la consistenza dei dati, occorre trovare un modo per gestire richieste eseguite da più servizi. La soluzione più comune è quella di usare il CQRS (*Command Query Responsibility Separation*) *pattern*: si tratta di affidare la responsabilità delle *query* e dei comandi a due rispettivi moduli in modo tale che venga garantita la consistenza dei dati durante le operazioni di lettura e scrittura [28].

Tuttavia, nell'ambito del progetto di questa tesi non è stato necessario implementare questo sistema perché è stata utilizzata solo un'istanza di un *database*. Nel progetto le informazioni da memorizzare e leggere non sono tante, introdurre ulteriori *database* sarebbe solo un costo in più e non porterebbe nessun vantaggio. Ad ogni modo, sono stati garantiti i principi di isolamento tra i dati dei diversi microservizi poiché i dati logicamente sono separati nell'unica istanza del *database* utilizzata.

4.1.3 Struttura

La figura 12 mostra l'architettura del *software* dopo il processo di migrazione. La decomposizione del monolite ha portato alla formazione di tre microservizi:

- Gitlab Service
- Slack Service
- User Service

Per lo sviluppo di ogni singolo servizio non si è seguito l'approccio dell'architettura stratificata vista per il *software* monolitico, ma si è deciso di seguire il *pattern* dell'architettura esagonale, tipico approccio usato per lo sviluppo di architetture a microservizi. Per questo motivo i microservizi, descritti più in dettaglio nei paragrafi successivi, sono organizzati in una parte più interna isolata dall'esterno che racchiude la

business logic del progetto e in un'altra parte che implementa gli *adapters*, cioè la parte visibile dall'esterno.

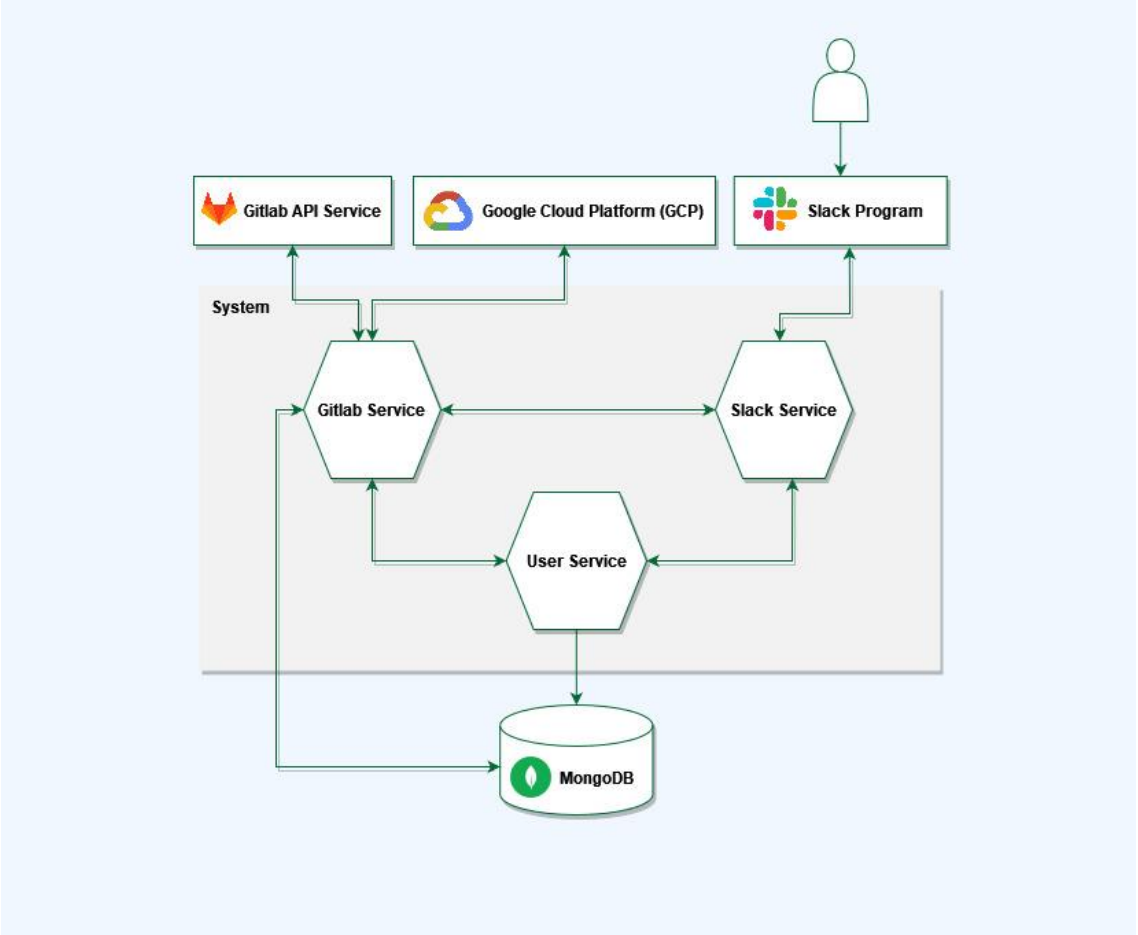


Figura 12 - Architettura del software dopo la migrazione da monolite a microservizi.

4.1.4 Gitlab Service

Il servizio chiamato "Gitlab Service" racchiude tutte le funzionalità relative ai *repository* Gitlab nei quali sono memorizzati i progetti aziendali. Per lo sviluppo è stato usato principalmente il *framework* NestJs e il linguaggio Typescript.

I moduli principali di questo progetto sono:

- *Adapters*: parte più esterna dove vengono inseriti i *controller*, cioè metodi che consentono di comunicare con il resto del sistema o con sistemi esterni. Per questo

servizio è stato creato un solo *adapter*, chiamato "Gitlab Adapter", nel quale sono stati implementati i *controller* che racchiudono le chiamate al servizio esterno di Gitlab. Si tratta quindi di *end-point* interni al sistema che al loro interno utilizzano le API di Gitlab per ricevere le informazioni necessarie.

- *Domain*: rappresenta la parte più interna che contiene i vari servizi; si tratta quindi della *business logic* del progetto. In particolare, il dominio di questo microservizio è composto da tutti i servizi e ciascuno implementa una singola funzionalità. I servizi implementati quindi sono i seguenti:
    - "Gitlab Default Setting": questo è il servizio che racchiude i metodi che si occupano di scaricare le informazioni dei progetti e gruppi di Gitlab e memorizzarle all'interno del *database*. In base ai dettagli eseguono gli aggiornamenti. Il servizio fa da *wrapper* dei singoli metodi descritti di seguito, ciascuno dei quali esegue una singola configurazione.
    - "Gitlab Protected Tags": all'interno di questo servizio ci sono i metodi che si occupano di creare i *protected tag* per ciascun progetto Gitlab. Per questa specifica funzionalità, le API di Gitlab non danno la possibilità di eseguire un aggiornamento nel caso in cui i *protected tags* già esistano. Per questo motivo, si è scelto di eliminarli e ricrearli per essere certi del funzionamento del servizio.
    - "Gitlab Protected Branches": per questo servizio, è stata seguita la stessa logica del servizio precedente dato che si tratta di due funzionalità molto simili.
    - "Gitlab Registry Tags": per eseguire la pulizia di questo registro è stato implementato un metodo chiamato "CleanRegistryTags" che utilizza l'API di Gitlab per poter cancellare le immagini Docker.
    - "Gitlab Clean Cache": così come per la pulizia del *container registry*, anche per questo servizio è stato implementato un metodo *ad hoc*. Per questo è stata utilizzata la libreria npm (*Node Package Manager*) di Google Cloud che offre i metodi già preconfigurati per eseguire operazioni sulla Google Cloud Platform sfruttando le API di Google. Per interagire con la piattaforma di Google è stato necessario generare un *service*

account per assegnare al servizio un'utenza autorizzata ad eseguire l'eliminazione dei dati contenuti dentro il *bucket*.

- "Gitlab Schedule": può essere visto come il servizio principale. Infatti, implementa i *Job*, cioè dei metodi che in modo automatico eseguono i metodi degli altri servizi.

Oltre a questi sono stati implementati anche altri due servizi chiamati rispettivamente "Gitlab Project" e "Gitlab Group" che si occupano di interrogare il *database* per memorizzare o scaricare informazioni dei progetti e dei gruppi del Gitlab aziendale.

In questo modo viene garantita anche a livello di progetto una separazione di logica che facilità la manutenibilità del *software* poiché permette la facile individuazione del codice: per ciascuna funzionalità corrisponde un metodo. Inoltre, dal punto di vista architetturale, adesso la *business logic* è isolata rispetto alle altre funzionalità esterne al dominio. Per questo è stato necessario introdurre i *Controller*, così da esporre e rendere disponibili agli altri servizi solo le funzionalità richieste.

Per eseguire le operazioni richieste, questo servizio deve comunicare sia con il *server* Gitlab sia con il *database*. Per quanto riguarda la comunicazione con il *server*, Gitlab nel proprio *storage* espone le informazioni utilizzando i DTO, *data transfer object*. In accordo ai principi di Typescript, quando si esegue una chiamata HTTP al *server* per recuperare i dati occorre definire come tipo di ritorno il formato del DTO. La stessa logica vale se l'obiettivo è quello di inviare dei dati al *server* Gitlab: vengono spediti con il formato che si aspetta il destinatario della richiesta.

Per quanto riguarda la connessione con il *database*, NestJs offre il *tool* Mongoose che fornisce una serie di metodi già implementati per creare la connessione con il *database*. In questo caso, per far comunicare correttamente *client* e *database*, sono state definite le *entities* per i progetti e i gruppi, cioè i tipi di dato che si aspetta di ricevere il *database*. Sulla base di queste informazioni, poi il servizio intero esegue gli aggiornamenti necessari.

Poiché è stato utilizzato il *pattern* dell'architettura esagonale, la *business logic* è distaccata dai formati di oggetto esterni: per questo, infatti, sono stati implementati dei *mapper* in

modo tale da distaccare e rendere indipendente la *business logic* dal servizio Gitlab. Questa caratteristica facilita il disaccoppiamento tipico dei microservizi, infatti, nel caso in cui Gitlab dovesse cambiare il proprio formato di dati, il servizio per poter continuare a funzionare necessità solo di una piccola modifica nel *mapper*, e non in tutto il progetto.

4.1.5 Slack Service

Questo servizio prende il nome dalle funzionalità che implementa; infatti, a questo servizio è associata quella parte di dominio relativa a tutte le funzionalità di Slack dando la possibilità all'utente di interagire con il resto del sistema sia per fornire dei *reminder* sia per notificare eventuali azioni automatiche che il *software* svolge in modo automatico.

Il passo preliminare per la creazione di questo microservizio è stato quello di creare una nuova applicazione tramite la *dashboard* fornita da Slack per gli sviluppatori. Dopo aver definito i dettagli dell'applicazione, vengono fornite le credenziali che permettono al microservizio di poter comunicare con le Slack API.

Dopo aver completato l'installazione della nuova *app* all'interno della *workspace* aziendale, è stata configurata la Socket Mode per gestire sia gli eventi sia le interazioni attraverso un WebSocket invece di utilizzare un *end-point* http. La *Socket Mode* di Slack prevede inoltre la configurazione delle *features* da abilitare al servizio al quale si collega. Nel caso di questo progetto, è stata abilitata la *Web Socket* per gestire le seguenti casistiche:

- Permettere all'utente l'esecuzione dei comandi tramite '\', rendendo il comportamento del *software* sempre più simile a quello tipico dei *bot*.
- Reagire agli eventi che si verificano nella *workspace* sfruttando le potenzialità delle *callback* offerte da Slack. Queste sono utilizzate per eseguire tutte quelle operazioni che devono essere eseguite quando si verifica un determinato evento. Per esempio, ogni qualvolta viene creato un nuovo canale occorre inserire anche il *bot*; tramite l'apposita *callback* si può gestire questo evento ed eseguire l'operazione di inserimento.

- Permettere l'interazione tramite dei modali o altri componenti grafici di Slack. Ciascun utente, infatti, tramite l'interfaccia della *home* del *bot* nella piattaforma Slack può lanciare i comandi desiderati.

Dopo aver completato questa configurazione lato applicazione, lo *step* successivo ha visto principalmente la migrazione del codice del *software* analizzato per l'applicativo monolite verso questo microservizio. Per questo, le tecnologie utilizzate sono rimaste invariate, quindi Typescript è il linguaggio di programmazione utilizzato il quale ha permesso di sfruttare le potenzialità di NodeJs e la libreria offerta da Slack. Come già detto durante la descrizione dei metodi per eseguire la decomposizione, questo microservizio è stato riadattato per consentire la comunicazione con gli altri servizi.

Anche questo servizio ha la necessità di creare una connessione con il *database* perché ciascun utente deve potersi configurare l'*access token* per eseguire le operazioni messe a disposizione da Gitlab Service. Quindi è necessario creare delle *entities* per consentire una comunicazione consistente.

4.1.6 User service

Lo User Service nasce dall'esigenza di dedicare un intero servizio alla gestione degli utenti. Ciascun utente per lanciare i comandi di configurazione dei progetti Gitlab tramite la piattaforma Slack deve prima fornire la propria identità tramite un *access token*.

La gestione di controllo dell'identità di ciascun utente viene affidata a questo servizio. Per ciascun utente occorre controllare se questo è in possesso di un *access token* e in caso di esito positivo deve essere concesso il permesso per eseguire la configurazione dei progetti accessibili da questo utente su Gitlab.

Per poter funzionare quindi deve interagire sia con lo Slack Service sia con Gitlab Service. Ha accesso diretto al *database* comune ai tre servizi dove vengono memorizzati gli *access token* dei vari utenti che configurano la propria utenza.

4.2 Processo di sviluppo

Uno dei vantaggi per cui è stata scelta l'architettura a microservizi è che lo sviluppo di ciascun servizio è molto veloce e consente di eseguire rilasci molto frequentemente. In un'ottica futura, nella quale il *bot* può espandersi ed integrare nuove funzionalità, i rilasci frequenti possono facilitare l'uso della metodologia *Agile*: essa prevede la creazione di piccoli *team* inter-funzionali, cioè, composti da membri provenienti da diverse aree di competenza, ciascuno con conoscenze e abilità specializzate. In questo modo, ciascun *team* è in grado di lavorare in maniera indipendente su ciascun servizio.

La struttura e l'organizzazione del processo di sviluppo del software è finalizzata ad utilizzare, insieme alla metodologia Agile, l'approccio chiamato DevOps, *Development Operations*: si tratta di un metodo di sviluppo che unisce il *team* dei programmatori con quello dei sistemisti. L'obiettivo principale di DevOps è superare le barriere tradizionali tra i *team* di sviluppo e di operazioni, promuovendo la collaborazione, l'automazione e l'integrazione continua durante tutto il ciclo di vita del *software*, dalla progettazione e lo sviluppo fino alla distribuzione e alla gestione operativa [29].

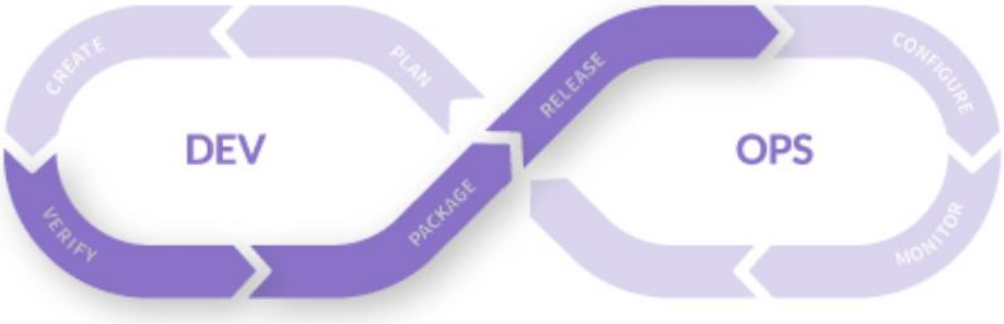


Figura 13 - Questo disegno rappresenta il flusso di lavoro end-to-end che coinvolge le diverse fasi del ciclo di vita del software, dall'ideazione e lo sviluppo fino alla distribuzione e alla gestione operativa.

La figura 13 rappresenta il flusso continuo delle sette fasi che caratterizzano il ciclo DevOps, chiamato comunemente "*DevOps Toolchain*" o "*DevOps Pipeline*". Di seguito sono elencate le attività che costituiscono questo ciclo:

- "Plan": è la prima attività preliminare della *toolchain*. In questa fase vengono definiti gli obiettivi da raggiungere, le priorità e le strategie per lo sviluppo, il rilascio e la gestione del *software*.
- "Create": è la fase in cui gli sviluppatori scrivono il codice sorgente per l'applicazione. Per gestire il codice e facilitare la collaborazione tra gli sviluppatori viene utilizzato Gitlab, già descritto e analizzato al paragrafo 3.2.2..
- "Verify": dopo l'implementazione il codice viene sottoposto ad una serie di *test* per verificarne la qualità e l'affidabilità.
- "Package": in questa fase viene generata una *build* del *software*. In particolare, si compila il codice mettendo insieme tutte le dipendenze necessarie per il corretto funzionamento, così da poter essere successivamente rilasciato. Per questo progetto di tesi ma anche per tutti i progetti a cui l'azienda lavora viene utilizzato Docker, come strumento per "pacchettizzare" un'applicazione.
- "Release": riguarda la fase in cui il *software* viene effettivamente installato e configurato nell'ambiente di produzione. L'automazione del *deployment* può coinvolgere l'utilizzo di strumenti di *deployment* continuo o l'orchestrazione di *container*.
- "Configure": durante questo *step* si gestisce l'ambiente di produzione in cui il *software* è stato rilasciato. Ciò implica la configurazione e il monitoraggio dei *server*, dei servizi e delle risorse necessarie per il funzionamento del *software* in produzione.
- "Monitor": dopo il *deployment*, il *software* viene costantemente monitorato per rilevare eventuali problemi o anomalie. Questo include il monitoraggio delle prestazioni, la registrazione degli errori e la raccolta di metriche per valutare l'utilizzo e il comportamento del sistema.

In questo modo, il *team* diventa inter-funzionale e la velocità con la quale si rilascia un *software* è maggiore rispetto ad un approccio classico.

Nei paragrafi successivi verrà spiegata la tecnica *Continuous Integration/Continuous Delivery*, utilizzata per automatizzare parte della *toolchain* DevOps con l'obiettivo di velocizzare l'intero flusso di operazioni. Successivamente verrà spiegato come sono state create le immagini Docker per i microservizi implementati.

4.2.1 Continuous Integration e Continuous Delivery

L'acronimo CI/CD fa riferimento all'unione di due tecniche, *Continuous Integration and Continuous Delivery*, utilizzate dalle aziende che utilizzano la metodologia DevOps per automatizzare e semplificare il processo di sviluppo a partire dai *commit* del codice dei singoli sviluppatori fino al rilascio del prodotto.

Nell'ambito di DevOps, il CI/CD svolge un ruolo cruciale per promuovere la collaborazione tra i *team* di sviluppo e le operazioni. Essa mira a rompere le barriere tra questi due gruppi, consentendo un flusso veloce e continuo di sviluppo, integrazione e distribuzione del *software*.

La *Continuous Integration* [30] è una pratica che fa riferimento ad una *pipeline* di operazioni che vengono eseguite ogni volta che viene fatto un *commit* nel *branch* in cui si caricano le nuove modifiche. La figura 14 illustra tutti gli *step* che riguardano in particolare l'integrazione continua. In particolare, per ogni *commit* e push viene eseguita la *pipeline* che crea la *build*, esegue gli *unit test* e analizza la qualità del codice. L'esito positivo di tutte le fasi dà al programmatore la possibilità di eseguire il *merge* sul *branch main* e di eliminare il *branch* sorgente, non più necessario. Per lo sviluppo di questo progetto, per ciascuna funzionalità da implementare è stato creato un *branch* e a partire da questo sono state utilizzate le tecniche della *Continuous Integration* offerte dal *tool* Gitlab.

All'interno della *pipeline* della CI è stato utilizzato anche il *tool* SonarQube: esso, in modo sempre automatico, esegue un'analisi del codice scritto in modo tale da offrire un *report* sulla qualità del codice identificando potenziali problemi, errori, vulnerabilità: tramite le sue indicazioni, i sviluppatori hanno l'opportunità di apportare eventuali modifiche per migliorare il codice appena caricato invitando anche ad utilizzare le *best practice* tipiche del linguaggio di programmazione utilizzato. L'utilizzo di questa eventuale fase permette di portare nel ramo principale di sviluppo codice che possiede già una corretta qualità del codice. In questo modo, si evita eventuale *refactoring* da pianificare dopo il rilascio del codice.

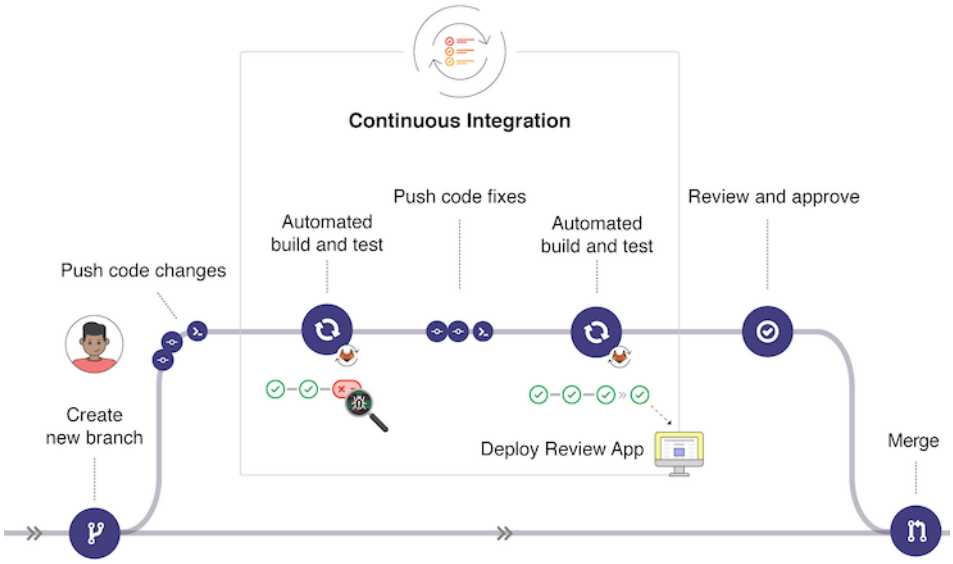


Figura 14 - Operazioni della Continuous Integration a partire dalla creazione di un nuovo branch fino al merge delle modifiche nel ramo di sviluppo principale.

La *Continuous Delivery* [31] consiste in una *pipeline* simile a quella della CI, il cui scopo è quello di creare automaticamente un pacchetto pronto per essere caricato in un ambiente di produzione. Questa tecnica permette la creazione automatica della *build* del *software*, cioè del pacchetto, solamente nel caso in cui le modifiche effettuate superino i *test* di integrazione e di qualità ad un ambiente di produzione, in modo che siano pronti per il rilascio.

La *continuous delivery* prevede la consegna del *software* in maniera graduale e progressiva, permettendo all'azienda di rilasciare rapidamente nuove funzionalità o correzioni di errori in modo controllato. Questo approccio promuove la flessibilità, la tempestività nel soddisfare le richieste dei clienti e la capacità di adattarsi ai mutamenti del mercato.

Nel caso del *software* analizzato, per ogni versione che è stata portata in produzione, è stata creata un'immagine Docker contenente una specifica versione del codice dell'applicazione. Successivamente, queste immagini Docker sono state memorizzate nel *container registry* associato al progetto.

In conclusione, l'utilizzo di queste due tecniche ha permesso di sviluppare in modo sicuro, cioè senza errori o *bug*, ciascun microservizio. L'esecuzione di ripetute volte delle *pipeline* sia durante la fase di sviluppo, sia durante la messa in produzione, ha permesso di eseguire dei controlli automatici e accurati grazie ai *test* e di avere *software* con codice ben scritto secondo le *best practice* delle tecnologie utilizzate.

4.2.2 Creazione Immagini Docker

Dopo la fase di sviluppo di ciascun microservizio, il processo di sviluppo ha previsto la creazione delle immagini Docker per dare la possibilità di creare i *container* per ciascun servizio. Come già descritto nel paragrafo dedicato alle tecnologie utilizzate, per poter creare un *container* si può definire un'immagine dalla quale si può avviare l'applicazione in qualsiasi macchina.

```
1  FROM node:17.0-slim
2
3  WORKDIR /usr/src/app
4
5  COPY package*.json ./
6  COPY ./dist ./dist/
7  COPY ./node_modules ./node_modules/
8  COPY .prod.env ./
9
10 EXPOSE 3000
11
12 CMD [ "npm", "run", "start:prod" ]
```

Figura 15 – Dockerfile che definisce i comandi necessari che servono per creare il container Docker.

In figura 15, si nota come è stato creato il Dockerfile per il servizio "Gitlab Service". Ecco una descrizione dei comandi utilizzati:

- FROM: indica l'immagine Docker di partenza dalla quale viene costruita la nuova immagine. In questo caso, si fa riferimento all'immagine di NodeJS della versione "17.0 – slim".

- WORKDIR: L'istruzione "WORKDIR /usr/src/app" imposta la *directory* di lavoro per tutte le successive istruzioni nel Dockerfile. Se la *directory* specificata non esiste, viene creata automaticamente.
- COPY: Questa istruzione viene utilizzata per copiare *file* o *directory* dal contesto di lavoro all'interno del *filesystem* del *container*. Per questo microservizio, questo comando è stato utilizzato per copiare all'interno della *directory* "/usr/src/app/" del container la *build* del progetto, le variabili di ambiente e tutte le dipendenze che servono per avviare il servizio.
- EXPOSE: L'istruzione "EXPOSE 3000" viene utilizzata per informare Docker che il *container* sarà in ascolto su specifiche porte. Questa istruzione serve principalmente a documentare le porte che il *container* rende disponibili per la comunicazione, ma non le rende effettivamente pubbliche.
- CMD: questa è l'ultima istruzione del Dockerfile e definisce il comando per avviare il servizio del *container*. Sostanzialmente l'istruzione "CMD" definisce il comando di *default* da eseguire all'avvio del *container*. In questo caso il comando che esegue è "npm run start:prod" che avvia il servizio utilizzando l'ambiente di sviluppo. In particolare, esegue lo *script* presente all'interno del file "package.json", importato all'interno della *directory* del *container* tramite il comando "COPY" descritto precedentemente.

In questo paragrafo è stato descritto il Dockerfile del servizio "Gitlab Service" poiché utilizza tecnologie molto simili e lo stesso *framework*, NodeJs, la struttura del progetto e l'immagine di partenza per la creazione del *container* risulta essere uguale. Per questo, infatti, le tre immagini risultano essere molto simili tra di loro.

5. Valutazione

In questo capitolo verrà eseguita un'analisi del *software* sviluppato. Dopo aver analizzato gli obiettivi raggiunti verranno descritti i casi d'uso testati per verificare il funzionamento del sistema. Successivamente, verranno valutate e confrontate le prestazioni, in termini di memoria e costi, tra la nuova versione e quella monolitica. Infine, viene evidenziato come l'architettura a microservizi riesce ad aumentare la robustezza e la resilienza del software monolite.

5.1 Obiettivi raggiunti

Grazie all'utilizzo dei microservizi, è stato possibile raggiungere gli obiettivi stabiliti. In particolare, allo *stack* tecnologico utilizzato nel monolite, NodeJs e Typescript, è stato aggiunto anche il *framework* NestJs rendendo quindi il sistema più moderno. "Gitlab Service", il microservizio che lo utilizza, risulta essere molto più performante poiché utilizza una tecnologia adatta alle funzionalità che implementa. Grazie all'utilizzo dei microservizi quindi è stato possibile aggiungere ulteriori funzionalità. Oltre a questo, essi danno la possibilità di creare eventuali nuovi servizi scegliendo la tecnologia che più si adatta alle esigenze richieste.

La scomposizione del monolite ha permesso di avere tre servizi, ognuno confinato nel proprio dominio. In questo modo, i tre servizi risultano essere isolati e responsabili di specifiche funzionalità. Questo aumenta la manutenibilità del *software*.

Inoltre, la decomposizione ha permesso di avere progetti ridimensionati rispetto all'unico progetto del monolite aumentando di conseguenza la velocità per eseguire eventuali

modifiche e aggiornamenti dei servizi esistenti; anche gli eventuali nuovi microservizi si possono agganciare molto velocemente al *bot* già in funzione.

Sicuramente questi obiettivi, raggiunti grazie all'utilizzo di un ambiente di sviluppo *cloud-based,* hanno contribuito ad avere a sua volta un *software* resiliente, cioè un sistema robusto di fronte ai guasti che possono succedere durante l'intero ciclo di vita del *software* mantenendo l'applicazione robusta e disponibile per gli utenti.

5.2 Casi d'uso

Un caso d'uso è sostanzialmente una sequenza di eventi che descrive una situazione specifica in cui il sistema viene utilizzato per raggiungere un obiettivo specifico. Ogni caso d'uso rappresenta un'interazione significativa tra un attore e il sistema, consentendo di identificare le azioni che l'attore compie e le risposte che il sistema fornisce.

Nell'ambito dell'Ingegneria del *Software*, una persona o a un'entità esterna al sistema che interagisce con il sistema stesso, scambiando informazioni con esso, viene definito attore: esso può fornire dati di *input* al sistema o ricevere dati di *output* dal sistema in risposta alle sue azioni o richieste.

Nel caso del progetto di questa tesi, poiché il *software* presenta anche degli automatismi, spesso il sistema reagisce ad eventi e non a richieste esplicite di un attore esterno. Di seguito verranno descritti i casi d'uso implementati per testare il funzionamento del sistema.

5.2.1 Configurazione automatica di un nuovo progetto

L'attore coinvolto in questo *use case* è solo il servizio esterno di Gitlab che si occupa di rispondere alle richieste che, in modo automatico, arrivano dal *software*. Lo scenario principale dello *use case* prevede che il dipendente durante le ore lavorative crea un nuovo progetto senza preoccuparsi delle configurazioni necessarie. Il *software*, durante le ore non lavorative, configura automaticamente il progetto, andando ad eseguire il flusso di operazioni implementate per la configurazione del progetto.

In figura 16 sono rappresentate le singole operazioni dello scenario principale:

- Il sistema lancia il flusso di operazioni per eseguire la sincronizzazione automatica delle informazioni dei progetti memorizzati sui *repository* remoti di Gitlab con il *database* del sistema.
- Gitlab, come attore esterno, esegue la configurazione dei progetti scelta dagli amministratori di sistema.
- Gitlab, dopo aver configurato le impostazioni del progetto, crea i *protected branches* e i *protected tags*.

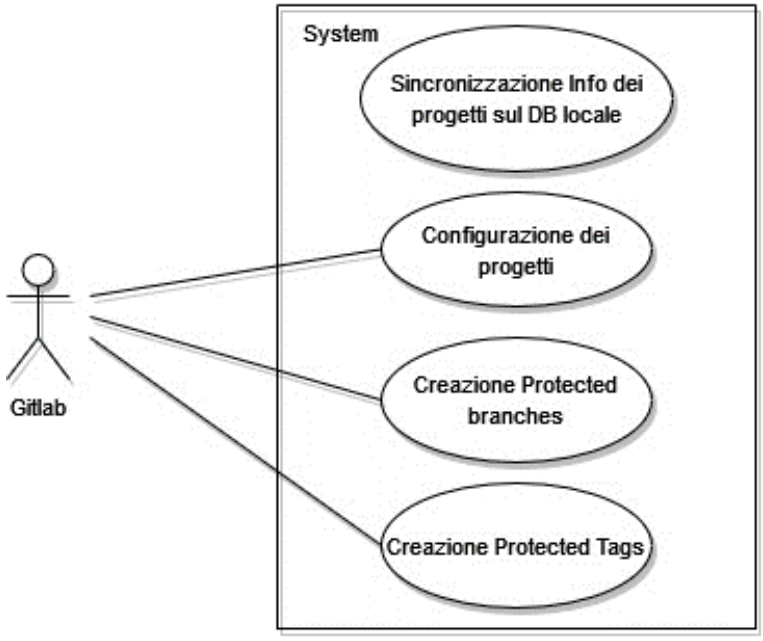


Figura 16 - Rappresentazione grafica dello use case della configurazione automatica dei progetti.

5.2.2 Configurazione manuale di nuovo progetto

In questo use case, gli attori sono gli utenti che possono configurare manualmente il proprio *access token* per autenticarsi al sistema esterno Gitlab per poter eseguire successivamente con la propria utenza le operazioni di configurazione. Ecco lo scenario di questo *use case*:

- L'utente deve creare un proprio *access token* da fornire al sistema per permettere l'autenticazione.
- L'utente apre la *home* del *chatbot* su Slack lancia il comando di autenticazione fornendo *l'access token* appena creato.

- L'utente dopo aver completato l'autenticazione lancia il comando per la configurazione del progetto.

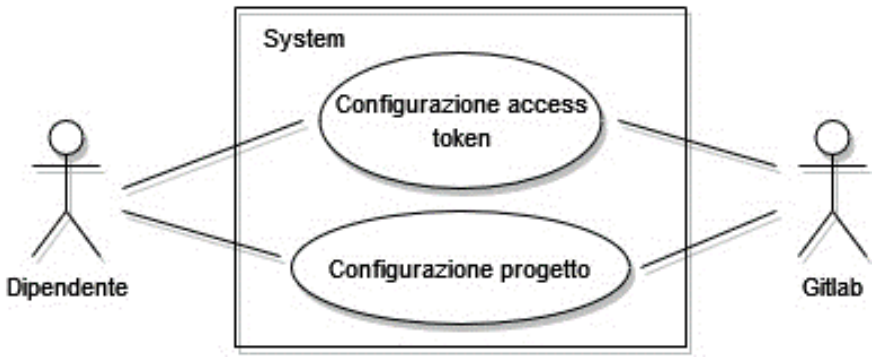


Figura 17 - Rappresentazione grafica dello use case relativo alla configurazione manuale del progetto.

5.2.3 Cancellazione automatica immagini Docker

Anche in questo caso d'uso l'unico attore coinvolto è Gitlab che essendo un sistema esterno gestisce le richieste che il *software* manda in automatico. In questo caso, è compito del *chatbot* eliminare le immagini del *Docker Registry* di ciascun progetto caricate su Gitlab. Il flusso principale delle operazioni è il seguente:

- Il *software* avvia l'esecuzione della funzionalità di eliminazione delle immagini del *Docker Registry*.
- Il *chatbot* interroga Gitlab per ottenere l'elenco dei progetti presenti.
- Per ogni progetto, accede al *Docker Registry* di Gitlab.
- Il *chatbot* identifica le immagini caricate nel *Docker Registry* per il progetto corrente ed elimina le immagini del *Docker Registry* per il progetto corrente.
- Il *software* passa al prossimo progetto e ripete i passaggi 4-5 fino a quando non ha elaborato tutti i progetti.

5.2.4 Ricezione notifiche reminder

La ricezione delle notifiche è uno *use case* con uno scenario che vede coinvolto l'utente che riceve in modo automatico dal *chatbot* il *reminder* per la compilazione settimanale,

sottomissione e approvazione del *timesheet*. In particolare, il flusso principale delle operazioni, rappresentato in figura 18, è descritto nei seguenti punti:

- Ogni venerdì lavorativo del mese: il sistema invia una notifica all'utente per ricordare di compilare il *timesheet* settimanale.
- Alla fine di ogni mese: il sistema invia una notifica a tutti i dipendenti per ricordare di approvare il *timesheet*.
- All'inizio di ogni mese: il sistema invia una notifica a chi di competenza per approvare i *timesheet* inviati nel mese precedente.

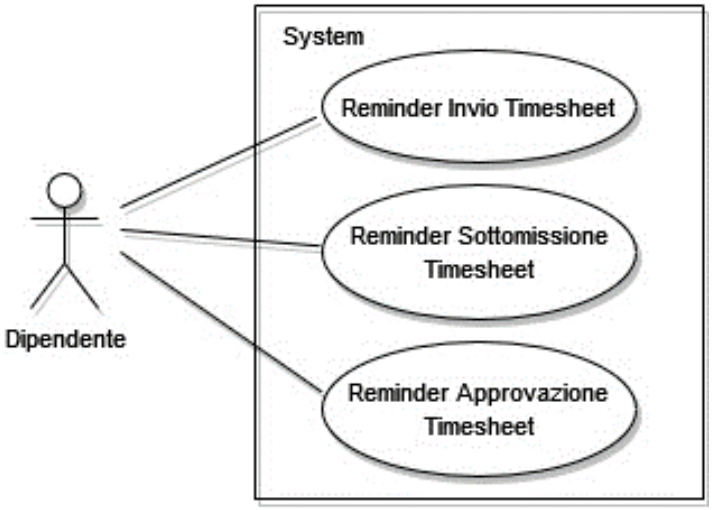


Figura 18 - Rappresentazione grafica dello use case con lo scopo di ricevere notifiche per la compilazione settimanale, sottomissione e approvazione dei timesheet.

5.2.5 Settaggio variabile d'ambiente

Questo caso d'uso descrive la situazione in cui l'azienda cliente richiede l'accesso all'istanza aziendale di Gitlab e richiede una configurazione diversa rispetto a quelle decise internamente.

L'attore coinvolto in questo caso d'uso è sia il dipendente sia Gitlab in quanto servizio esterno. Ecco il flusso delle operazioni principali di questo *use case*:

- L'azienda cliente richiede l'accesso all'istanza aziendale di Gitlab con una configurazione diversa da quella decisa internamente.

- Il dipendente, in qualità di responsabile della configurazione, si occupa di gestire la richiesta.
- Il dipendente modifica la variabile d'ambiente "bot ignore settings" e la imposta in modo appropriato per evitare la configurazione automatica del progetto come richiesto dall'azienda cliente.
- Il sistema rileva la modifica della variabile d'ambiente e, in base alla sua impostazione, evita la configurazione automatica del progetto in questione.

5.2.6 Cancellazione automatica cache

Per eseguire le *pipeline* degli sviluppi dei progetti viene utilizzato un *bucket* all'interno della Google Cloud Console che funge da *cache*. Lo scopo di questo *use case* è mostrare i passi che esegue il *bot* per cancellare il contenuto di questo *bucket*. Questa operazione viene eseguita, così come avviene per la cancellazione delle immagini Docker, la prima e terza domenica del mese. In questo caso, entra in gioco l'attore esterno che è Google Cloud che essendo contattato dal *software* si occupa di cancellare il contenuto del *bucket*.

Ecco i passi che vengono eseguiti nel contesto di questo *use case*:

- Il *bot* contatta Google Cloud per cancellare il contenuto del *bucket* nella *console* di Google Cloud.
- Google Cloud cancella il contenuto del *bucket* per le *pipeline* degli sviluppi dei progetti.
- Il *bot* conferma che il contenuto del *bucket* è stato cancellato.

5.3 Prestazioni

5.3.1 Memoria

La memoria necessaria per un *software* a microservizi rispetto ad un monolite dipende da tanti fattori. In questo contesto, il monolite da cui parte l'analisi contiene al suo interno

tutti i componenti, come l'interfaccia utente, la *business logic* e i moduli necessari per l'accesso al *database*. Di conseguenza, l'utilizzo della memoria è relativamente più contenuto rispetto alla nuova versione che utilizza i microservizi. La nuova versione, invece, con tre microservizi, richiede una quantità di memoria allocata maggiore rispetto alla singola istanza del monolite.

Inoltre, il monolite viene eseguito ogni volta all'interno di un singolo processo; quindi, tutti i moduli dei domini condividono lo stesso ambiente e quindi la stessa memoria. Naturalmente la memoria necessaria per permettere il funzionamento dipende dalle dimensioni dell'intero monolite. Per quanto riguarda invece il *software* dopo la migrazione, il sistema risulta essere suddiviso in componenti separati eseguiti in modo indipendente. Ciascun microservizio è un processo a parte e può richiedere una specifica quantità di memoria che può essere diversa da quella che richiedono gli altri servizi. Ovviamente i tre microservizi richiedono una quantità di memoria allocata maggiore rispetto alla singola istanza del monolite.

Occorre anche considerare il fatto che la comunicazione tra i vari servizi aumenta *l'overhead*; la comunicazione attraverso la rete tra i vari servizi prevede l'allocazione di ulteriori risorse facendo aumentare quindi la memoria complessiva utilizzata.

5.3.2 Costi

Il costo per migrare un *software* da una versione monolite ad una nuova versione che utilizza i microservizi dipende da diversi fattori. Innanzitutto, la struttura del progetto di partenza e la sua organizzazione possono influire sull'*effort* da erogare durante la fase di analisi e progettazione. È importante, infatti, che durante la prima fase venga analizzata per bene l'architettura esistente per comprendere come procedere con la decomposizione in microservizi.

Per quanto riguarda la fase di implementazione, gli strumenti e le tecnologie hanno un ruolo fondamentale nel costo complessivo. Nel contesto del progetto di tesi, la scelta di nuove tecnologie, come NestJs, ha comportato un costo maggiore in termini di *effort* perché è stata richiesto un'ulteriore analisi per capire come utilizzare e integrare questo *framework* all'interno del sistema.

Oltre ai costi relativi *all'effort*, la migrazione verso un sistema distribuito ha comportato un aumento del costo perché non si utilizza più una singola macchina per una sola istanza ma più macchine che devono essere anche in grado di comunicare tra di loro. Per questo motivo, infatti, il costo complessivo dell'infrastruttura comprende il costo di ciascuna macchina necessaria per ogni servizio. Inoltre, il sistema distribuito ha richiesto uno sforzo maggiore per configurare le risorse e l'ambiente di produzione per ciascun microservizio per garantire che il *deployment* funzioni correttamente.

Per quanto riguarda le risorse utilizzate da ciascun servizio, si riesce ad avere un'ottimizzazione dei costi. Kubernetes, *tool* utilizzato per il *deploy* dei microservizi, tramite *l'auto-scaling* riesce ad assegnare le risorse in base all'uso richiesto. In questo modo, ciascun servizio utilizza le risorse di cui necessità. Il costo, di conseguenza, risulta essere proporzionato all'uso. Questo, quindi, permette di avere un calo del costo nel momento in cui il servizio viene utilizzato di meno. Inoltre, *l'auto-scaling* riduce *l'effort* del *team* Operations, poiché non deve gestire manualmente la scalabilità delle macchine che eseguono i microservizi.

5.4 Parametri di valutazione

5.4.1 Scalabilità

Poiché i servizi dopo la migrazione sono ben isolati tra di loro, è possibile scalare le risorse in base alle richieste di ciascun servizio. Il *software* monolite, per sua natura, risulta essere difficilmente scalabile perché nel caso di un aumento di risorse da parte del sistema occorre scalare l'intero sistema impattando quindi sia sulla memoria richiesta sia sui costi. Invece, i microservizi permettono di avere una maggiore granularità nell'approccio alla distribuzione delle risorse.

È possibile ottimizzare l'utilizzo delle risorse poiché non è necessario scalare l'intero sistema quando solo una parte specifica richiede maggiori risorse. Inoltre, grazie all'utilizzo di Kubernetes, che offre *l'auto-scaling* è possibile configurare automaticamente le risorse disponibili in base alle esigenze del sistema. In questo modo

si garantisce anche una certa continuità nel funzionamento del sistema evitando quindi eventuali spegnimenti di parti del *software* per mancate risorse allocate.

5.4.2 Affidabilità

L'architettura a microservizi spesso viene scelta per aumentare la resilienza del *software*. Nel caso di questo progetto di tesi, grazie anche all'isolamento di ciascun servizio, se uno di questi dovesse fallire o riscontrare problemi, gli altri servizi possono continuare a funzionare in modo indipendente.

Per esempio, se il "Gitlab Service" per qualche motivo non dovesse più funzionare e quindi si spegne, gli altri due servizi rimarrebbero attivi, e quindi continuerebbe a funzionare lo "Slack Service" senza essere intaccato dal malfunzionamento dell'altro servizio.

In conclusione, quindi, lo sviluppo indipendente di un servizio specifico ha permesso di non intaccare il funzionamento degli altri servizi riducendo quindi la probabilità di avere malfunzionamenti.

5.4.3 Manutenibilità

La manutenibilità del codice dopo la migrazione è aumentata notevolmente perché con i microservizi è più semplice apportare modifiche ad un singolo componente dell'applicazione senza influire sul funzionamento degli altri. Infatti, è possibile eventualmente modificare un solo servizio o correggere un *bug* in un servizio evitando che gli altri vedano completamente le modifiche apportate. Quando le modifiche saranno pronte per essere rilasciate, non bisogna preoccuparsi di distribuire l'intera applicazione: è sufficiente distribuire il microservizio modificato. Questo riduce il rischio di regressione, ovvero il rischio che una modifica introduca errori o problemi nel *software* che funzionava correttamente in precedenza. Inoltre, semplifica la gestione dei cambiamenti nel codice aumentando quindi la manutenibilità dell'intero sistema.

5.4.4 Qualità del codice

Nella valutazione del *software* dopo il *refactoring*, è emerso che la nuova versione rispetto alla precedente possiede una qualità del codice molto simile alla versione monolitica. In particolare, in entrambe le versioni sono stati utilizzati i *tool* ESLint e Prettier, strumenti che insieme possono contribuire notevolmente a garantire la qualità del codice. Entrambi sono ampiamente utilizzati dagli sviluppatori per migliorare la leggibilità, la manutenibilità e l'omogeneità del codice.

ESLint è uno strumento di analisi statica del codice che aiuta a individuare errori, pratiche non consigliate e potenziali problemi nel codice sorgente [32]. Può essere configurato con regole personalizzate o predefinite per adattarsi alle specifiche esigenze del progetto. ESLint offre un'ampia varietà di regole, che vanno dal controllo della sintassi all'indentazione corretta, dalle *best practice* per la gestione degli errori alla conformità agli *standard* di codifica. ESLint nasce come strumento di analisi per codice Javascript. Tuttavia, tra i *plugin* configurabili, è possibile aggiungere il *parser* di Typescript che ha permesso di poter utilizzare questo strumento anche nel progetto di tesi.

Prettier, d'altra parte, si concentra sulla formattazione automatica del codice per rendere uniforme lo stile di scrittura. Impone un insieme predefinito di regole di formattazione, che comprendono l'indentazione, gli spazi bianchi, le parentesi graffe, la punteggiatura e altri aspetti [33]. Prettier si occupa solo della formattazione e non effettua controlli statici come ESLint, ma entrambi possono essere integrati insieme per ottenere il massimo beneficio. È possibile utilizzare Prettier per diversi linguaggi di programmazione, tra cui Typescript.

La combinazione di ESLint e Prettier può essere configurata in modo da lavorare in sinergia. ESLint viene utilizzato per garantire la correttezza sintattica e semantica del codice, nonché per individuare errori e problemi potenziali, mentre Prettier si occupa della formattazione automatica del codice secondo le regole stabilite.

Questa combinazione ha aiutato a mantenere uno stile di codifica coerente e a ridurre gli errori umani, poiché i membri del *team* seguono automaticamente le regole di formattazione e le *best practice* stabilite. Inoltre, ha facilitato il processo di revisione del

codice, in quanto è possibile concentrarsi su aspetti più importanti come la logica e l'efficienza, sapendo che l'aspetto formale è gestito automaticamente.

In conclusione, utilizzando ESLint insieme a Prettier, è stato possibile migliorare la qualità del codice, promuovendo uno stile di scrittura uniforme e individuando potenziali problemi nel codice, migliorando così la leggibilità, la manutenibilità e l'affidabilità del *software*.

6. Conclusioni

L'obiettivo di questa tesi è stato quello di dimostrare come la migrazione di un *software* monolitico verso una nuova architettura a microservizi sia una scelta che rende l'applicativo più robusto ed efficiente.

La suddivisione del monolite in tre microservizi ha permesso innanzitutto di scegliere le tecnologie più performanti per i servizi implementati. Per lo sviluppo *software* da cui è nato questo progetto invece era stato scelto un *stack* tecnologico fisso unico per i domini analizzati. La scomposizione ha permesso di introdurre quindi tecnologie più moderne, ad esempio NestJs, e specializzate nelle funzionalità del singolo servizio, dando grande flessibilità agli sviluppi futuri.

Inoltre, ha reso più semplice l'organizzazione dell'applicativo: nel monolite i tre domini si intrecciavano senza nessun tipo di divisione creando quindi un unico blocco rigido rispetto alla gestione di eventuali modifiche. Una piccola modifica potenzialmente poteva richiedere tanto tempo perché le parti logicamente separate si trovavano in un unico blocco fisico, oltre che introdurre il rischio di *bug* o di regressioni.

Infine, la migrazione ha reso l'assistente virtuale più flessibile e potenzialmente più utile perché questa nuova architettura facilità e velocizza l'introduzione di nuove funzionalità. In questo modo si presta bene ad assumere nuovi ruoli e responsabilità quando nascono nuove necessità.

6.1 Sviluppi futuri

Il progetto di tesi nasce per rendere il *software* un vero assistente virtuale. Per questo, uno dei possibili sviluppi futuri è quello di utilizzare l'AI, *Artificial Intelligence*, per renderlo un *Natural Language Processing* (NLP) *bot*, cioè un *chatbot* in grado di elaborare le domande poste dall'utente ed eseguire delle operazioni sulla base delle parole presenti nella domanda. In particolare, si vuole addestrare il *bot* in modo tale che sia in grado di rispondere alla richiesta di creazione di un nuovo progetto: l'obiettivo deve essere quello eseguire lo *startup* automatico e di configurare le impostazioni relative al processo di sviluppo. Al momento, lo *startup* del progetto viene eseguito manualmente dall'utente e oltre a questo deve configurare strumenti di sviluppo, qualità del codice e altre impostazioni. Per questo motivo, è possibile anche che le configurazioni risultino diverse tra i vari progetti. La creazione di un sistema intelligente in grado di recepire subito la richiesta dell'utente permette in questo caso di ridurre notevolmente i tempi di creazione di un *boilerplate* per un nuovo progetto.

Inoltre, si vuole anche integrare il *bot* con Terraform, uno strumento *open source infrastructure as code*: viene utilizzato per creare e gestire l'infrastruttura di un *software* [34]. In particolare, si vuole che in base alla richiesta effettuata in *chat* dall'utente, il *bot* interagisca con Terraform per definire automaticamente l'infrastruttura del progetto appena creato.

L'integrazione del *bot* con Terraform consentirebbe di automatizzare il processo di *provisioning* dell'infrastruttura in risposta alle richieste degli utenti. Ad esempio, supponiamo che un utente chieda al *bot* di creare un nuovo progetto. Il *bot* può ricevere questa richiesta dalla *chat* e utilizzare Terraform per creare dinamicamente le risorse necessarie per supportare il progetto, come ad esempio un'istanza di *server*, un database, un bilanciamento del carico, ecc..

Attraverso l'integrazione, il *bot* può interagire con Terraform in modo da poter generare automaticamente i file di configurazione specifici per il progetto in base alle preferenze dell'utente. Questi *file* di configurazione di Terraform definiranno le risorse da creare, le loro proprietà e le dipendenze tra di loro.

Una volta che il *bot* ha generato i *file* di configurazione di Terraform, può avviare automaticamente il processo di esecuzione di Terraform per creare l'infrastruttura specificata. Terraform si occuperà quindi di comunicare con il *provider cloud* (come ad esempio AWS, Azure o Google Cloud) per creare e configurare le risorse richieste.

Questa integrazione permette di automatizzare il processo di creazione e gestione dell'infrastruttura, eliminando la necessità di eseguire manualmente comandi o interagire direttamente con i *provider cloud*. Inoltre, offre la possibilità di mantenere l'infrastruttura come codice, consentendo una gestione dell'ambiente di sviluppo versionata e riproducibile.

Un enorme vantaggio dall'uso di Terraform è la riduzione del tempo, e di conseguenza dei costi, necessari per effettuare un *deploy*. Un *software* composto da tanti microservizi richiede tanto tempo per essere *deployato* nell'ambiente di produzione. Con Terraform, invece, i tempi si accorciano notevolmente; questo, quindi, consente di far aumentare la produttività al *team* che si occupa dell'infrastruttura.

In sintesi, l'integrazione del *bot* con Terraform consente di definire automaticamente l'infrastruttura del progetto appena creato in base alle richieste degli utenti. Questo approccio automatizzato semplifica e velocizza il processo di *provisioning* dell'infrastruttura, migliorando l'efficienza e la consistenza nell'implementazione delle risorse richieste.

Bibliografia

[1] L. De Lauretis, «From Monolithic Architecture to Microservices Architecture,» in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019.

[2] F. Ponce, G. Márquez e H. Astudillo, «Migrating from monolithic architecture to microservices: A Rapid Review,» in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, 2019.

[3] C. Richardson, Microservices Patterns with examples in Java, Manning Pubblications Co., 2019.

[4] M. Waseem, P. Liang e M. Shahin, «Software Architecture Design of Microservices Systems,» 2022.

[5] S. Newman, Building Microservices, 2015.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, «Introduction to Algorithms,» 2009.

[7] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou e Z. Li, «Microservices: architecture, container, and challenges,» in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2020.

[8] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin e L. Safina, «Microservices: yesterday, today, and tomorrow,» June 2016.

[9] F. Suprapto, K. Surendro e W. Sunindyo, «Circuit Breaker in Microservices: State of the Art and Future Prospects,» *IOP Conference Series: Materials Science and Engineering,* vol. 1077, p. 012065, February 2021.

[10] F. Tapia, M. Á. Mora, W. Fuertes, H. Aules, E. Flores e T. Toulkeridis, «From Monolithic Systems to Microservices: A Comparative Study of Performance,» *Applied Sciences,* vol. 10, 2020.

[11] NestJs, «NestJs Docs,» [Online]. Available: https://docs.nestjs.com/.

[12] NodeJs, «Introduction to NodeJs,» [Online]. Available: https://nodejs.dev/en/learn/introduction-to-nodejs/.

[13] Wikipedia, *Node.js — Wikipedia, L'enciclopedia libera,* 2022.

[14] S. Overflow, «Stack Overflow Developer Survey,» [Online]. Available: https://survey.stackoverflow.co/2022/#section-most-popular-technologies-web-frameworks-and-technologies.

[15] Wikipedia, *MongoDB — Wikipedia, L'enciclopedia libera,* 2023.

[16] DB-Engines, *DB-Engines Ranking,* 2023.

[17] MongoDB, «What is MongoDB,» [Online]. Available: https://www.mongodb.com/it-it/what-is-mongodb.

[18] Docker, «Docker Docs,» [Online]. Available: https://docs.docker.com/.

[19] B. Bashari Rad, H. Bhatti e M. Ahmadi, «An Introduction to Docker and Analysis of its Performance,» *IJCSNS International Journal of Computer Science and Network Security,* vol. 173, p. 8, March 2017.

[20] Kubernetes, «What is Kubernetes?,» [Online]. Available: https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/.

[21] G. Cloud, «Che cos'è Kubernetes?,» [Online]. Available: https://cloud.google.com/learn/what-is-kubernetes.

[22] Git, «Git,» [Online]. Available: https://git-scm.com/docs.

[23] Slack, «Apis Slack,» [Online]. Available: https://api.slack.com/apis.

[24] Gitlab, «Develop with GitLab,» [Online]. Available: https://docs.gitlab.com/ee/api/.

[25] Slack, «Intro to Socket Mode,» [Online]. Available: https://api.slack.com/apis/connections/socket.

[26] Gitlab, «Webhooks,» [Online]. Available: https://docs.gitlab.com/ee/user/project/integrations/webhooks.html.

[27] M. Štefanko, O. Chaloupka e B. Rossi, «The Saga Pattern in a Reactive Microservices Environment,» 2019.

[28] J. Kabbedijk, S. Jansen e S. Brinkkemper, «A case study of the variability consequences of the CQRS pattern in online business software,» July 2012.

[29] P. a. S. R. a. P. I. Perera, «Improve software quality through practicing DevOps,» *2017 Seventeenth International Conference on Advances in ICT for Emerging Regions (ICTer),* p. 14, 2017.

[30] Gitlab, «Continuous Integration,» [Online]. Available: https://docs.gitlab.com/ee/ci/introduction/index.html#continuous-integration.

[31] Gitlab, «Continuous Delivery,» [Online]. Available: https://docs.gitlab.com/ee/ci/introduction/#continuous-delivery.

[32] ESLint, «Documentation - ESLint,» [Online]. Available: https://eslint.org/docs.

[33] Prettier, «What is prettier?,» [Online]. Available: https://prettier.io/docs/en/index.html.

[34] Terraform, «Documentation Terraform,» [Online]. Available: https://developer.hashicorp.com/terraform/docs.

[35] A. Bucchiarone, N. Dragoni, S. Dustdar, S. Larsen e M. Mazzara, *From Monolithic to Microservices: An experience report,* 2017.

[36] Typescript. [Online]. Available: https://www.typescriptlang.org/.