Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Triennale Informatica

La transizione da architettura monolitica a microservizi

un caso studio nel settore del turismo

Relatore: Prof. Marco Viviani

Relazione della prova finale di:
J. Yang Xiang
Matricola: 866013

Anno Accademico 2022-2023

Ai miei genitori, fratelli, amici,
per aver contribuito a plasmare la persona che sono diventato oggi.

Ai miei compagni CIP,
per essermi stato accanto lungo tutto il percorso sin dall'inizio.

A me stesso,
per aver affrontato ogni difficoltà senza mai arrendermi.

Indice

1 Introduzione

L'architettura **monolitica** è una delle prime e più semplici forme di architettura software. Questo approccio prevede l'utilizzo di un **singolo componente**, ovvero un unico sistema che gestisce tutte le funzionalità dell'applicazione, senza alcuna suddivisione in moduli o componenti indipendenti. Questo sistema, noto come **monolite**, viene sviluppato, testato e distribuito come un'entità unica. Questa struttura è stata per molto tempo la soluzione preferita per lo sviluppo di applicazioni software, poiché è relativamente semplice da implementare e gestire. Tuttavia, con l'aumento della **complessità** delle applicazioni e la necessità di **scalabilità**, **affidabilità** e **manutenibilità**, questo approccio ha mostrato alcune limitazioni. Ad esempio, la gestione di un sistema monolitico può diventare difficoltosa quando si verificano problemi di **prestazioni** o quando si desidera introdurre nuove funzionalità nell'applicazione. Inoltre, in caso di necessità di **scalabilità orizzontale**, ovvero l'espansione della capacità dell'applicazione mediante l'aggiunta di nuovi server, può risultare problematico scalare l'intero sistema monolitico, poiché tutti i componenti dell'-applicazione sono intimamente legati tra loro. Nonostante queste limitazioni, l'architettura monolitica rimane ancora oggi una solu-zione valida per alcune tipologie di applicazioni, soprattutto per quelle di piccole dimensioni o con bassa complessità funzionale. Tuttavia, queste nuove esigenze si è sviluppata una nuova forma di architettura, quella a **microservizi**, che ha preso sempre più piede negli ultimi anni.

Questa tesi si propone quindi di esplorare il processo di transizione dall'architettura monolitica all'architettura a microservizi, analizzando le **sfide**, i **benefici** e uno **studio del caso** su una azienda che sta preparando tale transizione. L'architettura a microservizi ha acquisito importanza negli ultimi anni grazie alla sua capacità di affrontare le problematiche riscontrate nelle architetture monolitiche. Mentre quest'ultime offrono semplicità nello sviluppo iniziale e nella comprensione, presentano come precedentemente accennato **limitazioni** in termini di scalabilità, rilascio continuo e gestione del codice. Al contrario, i microservizi offrono vantaggi come il *reduced coupling*, *single responsibility*, scalabilità, diversità tecnologica e una gestione dei dati decentralizzata, consentendo alle organizzazioni di innovare e ottenere un vantaggio competitivo. Nonostante ciò, l'adozione dei microservizi presenta comunque una serie di sfide, tra cui la complessità dei sistemi distribuiti, la comunicazione tra i servizi, la difficoltà di testing, gli ostacoli nel rilascio in produzione e le implicazioni organizzative.

1.1 Obiettivi della ricerca

Questa tesi valuterà i fattori che influenzano la decisione di passare dall'architettura monolitica all'architettura con microservizi, con un focus su una azienda che chiameremo *Travel Company* che ha raggiunto una certa importanza nel mercato dei viaggi e possiede una comprensione approfondita del suo panorama aziendale. Attraverso l'analisi del **caso di studio**, le *best practices* dell'industria e le opinioni degli esperti, questa ricerca mira a fornire un quadro completo per consentire alle organizzazioni di navigare con successo nel processo di transizione, affrontare le sfide e sfruttare i benefici offerti dall'architettura a microservizi.

1.2 Presentazione capitoli

Il Capitolo 2, **"Background e motivazione"**, rappresenta la base per la comprensione delle due architetture software in oggetto. In particolare, esso si propone di definire con precisione le differenze tra le due architetture e di evidenziarne i rispettivi vantaggi. In questo modo, si intende mettere in luce le ragioni che spingono le aziende a valutare la migrazione verso un'architettura a microservizi, nonché a comprenderne i punti di forza e di debolezza rispetto ai monoliti.

Il Capitolo 3, **"Tecniche per la transizione"**, descrive le tecniche e le strategie che possono essere adottate per facilitare il processo di transizione. In particolare, vengono presentate le soluzioni per la decomposizione del sistema monolitico in microservizi, per la gestione delle comunicazioni tra i servizi, per la gestione del deployment e delle operazioni di sistema. L'obiettivo di questo capitolo è quello di offrire una panoramica completa sulle soluzioni tecnologiche che possono essere utilizzate per rendere il processo di migrazione più fluido ed efficace.

Il Capitolo 4 **"Un caso di studio nell'ambito del turismo"**, rappresenta il cuore della tesi e dà il via all'analisi del caso studio della Travel Company che ha proposto la transizione; ci si concentrerà sulla descrizione di questo processo in corso, presentando gli obiettivi dell'azienda, le sfide che sta affrontando e le soluzioni che sta considerando per la transizione. Si parlerà della loro architettura corrente e di quella concettuale che si ha in obiettivo di avere alla fine della transizione, andando in dettaglio sui principi architetturali e le sue caratteristiche.

Il Capitolo 5 **"Approccio progettuale"**, andrà in dettaglio nel design proposto dall'azienda, analizzando il suo approccio ai limiti (*boundaries*) dei servizi, l'identificazione di componenti che possono essere trasformati in microservizi, i loro meccanismi di comunicazione, la gestione dei dati, la sicurezza - in cui si toccheranno punti come l'autenticazione, la crittografia e il *logging/monitoring*, la strategia del *Continuous Delivery and Deployment*.

Il Capitolo 6 **"Soluzioni di *eCommerce*"**, parlerà della scelta dell'azienda sulla piattaforma da adottare, tra uno di terze parti oppure l'implementazione di una personalizzata ad-hoc, analizzando in dettaglio i loro vantaggi e svantaggi.

Il Capitolo 7 **"Infrastruttura tecnologica"**, toccherà infine i vari punti chiave che riguardano l'eventuale piattaforma personalizzata presentata nel capitolo precedente, presentando le varie sezioni, come la rete pubblica, quella privata, i strumenti di rilascio e monitoraggio usati.

Questo caso studio offrirà un'opportunità per comprendere come un'azienda si sta preparando per la transizione da un'architettura monolitica a una a microservizi per esaminare le scelte che dovrà fare in questo processo di cambiamento.

2 Background e motivazione

Questo capitolo contiene l'introduzione e la motivazione della trasformazione dalla architettura monolitica a quella a microservizi. Vengono descritte entrambe le architetture, monolitica e a microservizi, e confrontate tra loro, presentando anche i casi d'uso in cui sono applicabili. L'obiettivo principale di questa sezione è fornire gli strumenti per comprendere le sfide che vengono risolte e il motivo per cui si sta attuando questa trasformazione.

2.1 Motivazione

Nel panorama tecnologico attuale, le aziende sono sempre più impegnate nella gestione di infrastrutture complesse e devono far fronte alla sfida di implementazioni e correzioni sempre più rapide e frequenti dei propri progetti. Inoltre, la richiesta di flessibilità e scalabilità è diventata sempre più pressante, tanto che molte organizzazioni stanno spostando i loro server da locale a cloud. In questo contesto, l'architettura a microservizi si presenta come una soluzione ideale per affrontare queste sfide e soddisfare le esigenze delle aziende moderne.

Oltre alla necessità di flessibilità e scalabilità, ci sono altri motivi per cui le organizzazioni stanno considerando la transizione:

1. **Flessibilità tecnologica**: con un'architettura monolitica, tutti i componenti dell' applicazione devono essere sviluppati utilizzando lo stesso linguaggio di programmazione e i medesimi framework, portandolo a limitare le scelte tecnologiche e rendere difficile adottarne di nuove. Con un'architettura a microservizi, ogni servizio può essere sviluppato utilizzando il linguaggio di programmazione e il framework più adatti per quello specifico servizio.

2. **Separazione dei repository**: i microservizi permettono una semplificazione maggiore della gestione del codice e la manutenzione del software in quanto ogni servizio ha il proprio repository, il che può semplificare la gestione del codice e rendere più facile la manutenzione del software. Con un monolita, l'intero codice dell'-applicazione è contenuto in un unico repository, su cui vanno effettuate tutte le modifiche.

3. **Resilienza ai fallimenti**: Il rischio di fallimento è ridotto in un'architettura a microservizi: in un monolita, un singolo errore può causare il fallimento dell'intera applicazione mentre con i microservizi, i singoli servizi possono fallire senza influire sul resto dell'applicazione. Inoltre, i servizi a cui viene rilevato un problema possono essere facilmente isolati e gestiti senza influire sul resto del sistema.

4. **Sicurezza**: a causa della natura centralizzata e all'interconnessione dei componenti all'interno di un monolite, l'ambiente di esecuzione è generalmente condivisa con un accesso agli stessi dati e risorse; ciò significa che se un componente viene compromesso da un attacco, potrebbe avere accesso a tutto l'ambiente e ai dati sensibili dell'applicazione. Dal lato dei microservizi, i componenti sono separati e indipendenti: un attacco che colpisce un singolo servizio avrà un impatto limitato sul resto del sistema. Meccanismi utilizzati nei microservizi comprendono autenticazione, autorizzazione e controlli di accesso; ogni servizio può avere le proprie

politiche di sicurezza e i meccanismi di difesa per proteggere i dati e le risorse che gestisce, permettendo una maggiore granularità nel controllo accessi e gestione di autorizzazioni.

In sintesi, l'architettura a microservizi offre numerosi vantaggi rispetto all'architettura monolitica, tra cui maggiore flessibilità, maggiore scalabilità, maggiore facilità di manutenzione e maggiore sicurezza. Tuttavia, la transizione a un'architettura a microservizi richiede una pianificazione attenta e una gestione adeguata dei servizi distribuiti.

2.2 Architettura monolitica

L'architettura monolitica rappresenta uno dei paradigmi più tradizionali nello sviluppo di software e ha svolto un ruolo di primaria importanza nella storia dell'informatica. Essa si contrappone alle moderne architetture distribuite e microservizi, che hanno guadagnato popolarità negli ultimi anni. Tuttavia, l'architettura monolitica continua ad essere utilizzata in molti contesti e offre vantaggi che la rendono ancora rilevante nel panorama attuale.

Immaginiamo di costruire una casa: potremmo iniziare con un progetto che prevede la realizzazione di fondamenta solide, una struttura portante robusta e diversi ambienti che si integrano in modo coerente. In pratica, stiamo costruendo un'unica entità, un monolito che rappresenta l'intera casa. In modo simile, l'architettura monolitica nel contesto dello sviluppo software implica la creazione di un'applicazione che funziona come un'unità coesa e indivisibile.

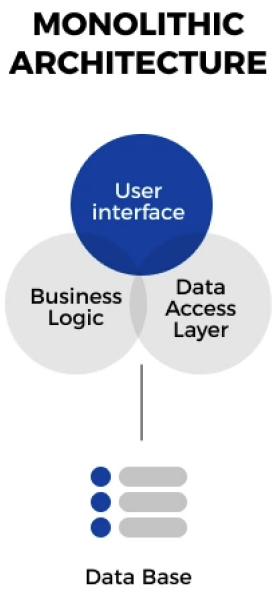

Figura 1: Schema di un'architettura monolitica, i componenti sono integrati in un unico sistema

In una architettura monolitica, illustrata in Figura 1, tutte le componenti dell'applicazione – il frontend, il backend, la logica di business e il database – sono raggruppate all'interno di un'unica entità. Ciò significa che tutte le funzionalità dell'applicazione sono implementate all'interno dello stesso codice sorgente e vengono eseguite sullo stesso processo o macchina virtuale.

Uno dei principali vantaggi di questa architettura è la **semplicità**. Poiché l'intera applicazione è implementata all'interno di un singolo monolito, non è necessario gestire la comunicazione e la sincronizzazione tra diverse componenti, come avviene con architetture distribuite. Questo semplifica lo **sviluppo**, il **deployment** e la **manutenzione** dell'applicazione.

L'architettura monolitica è spesso più facile da testare rispetto a sistemi distribuiti: poiché tutte le funzionalità sono integrate, è possibile eseguire test end-to-end per verificare il corretto funzionamento dell'applicazione nel suo complesso. Le librerie e gli strumenti di test per le applicazioni monolitiche sono inoltre ampiamente disponibili e consolidati nel tempo.

Questa architettura presenta tuttavia anche alcune limitazioni; a causa della sua natura monolitica, l'applicazione può diventare complessa e difficile da scalare. L'aggiunta di nuove funzionalità potrebbe richiedere modifiche a una porzione significativa del codice sorgente, aumentando il rischio di errori e la complessità complessiva del sistema.

Infine, l'architettura monolitica potrebbe soffrire di un'alta dipendenza tra le diverse parti dell'applicazione in quanto un errore in una porzione del codice potrebbe avere un impatto negativo sull'intero sistema. La manutenzione e l'evoluzione dell'applicazione possono diventare problematiche a lungo termine, specialmente se il codice non è ben strutturato.

2.3 Architettura a microservizi

Passiamo ora a parlare dell'architettura a microservizi, un approccio moderno allo sviluppo di software che ha guadagnato sempre più popolarità negli ultimi anni. Rispetto alla architettura monolitica di cui abbiamo parlato in precedenza, l'architettura a microservizi presenta una serie di caratteristiche distintive che offrono vantaggi significativi in termini di scalabilità, resilienza e sviluppo agile.

Immaginiamo ora di costruire una città invece di una casa. Invece di un unico edificio massiccio, la città è composta dai vari edifici più piccoli, ognuno con la propria funzione specifica. Questi edifici possono essere costruiti e modificati indipendentemente l'uno dall'altro, ma lavorano insieme per formare un ecosistema interconnesso. L'architettura a microservizi applica lo stesso concetto all'ambito dello sviluppo software.

In un'architettura a microservizi, come illustrato in Figura 2, l'applicazione viene suddivisa in un insieme di servizi autonomi, ognuno dei quali rappresenta un modulo specifico con una funzionalità ben definita. Ogni servizio opera come un'entità indipendente e può essere sviluppato, testato, distribuito e scalato in modo autonomo. Questi servizi comunicano tra loro attraverso meccanismi di messaggistica, come API o eventi.

Un vantaggio chiave dell'architettura a microservizi è la scalabilità. Poiché ogni servizio è indipendente, è possibile scalare singoli servizi in base alle esigenze specifiche, senza dover scalare l'intera applicazione. Questo approccio consente di gestire carichi di lavoro elevati e di adattarsi in modo flessibile alle richieste dell'utente.

Si promuove inoltre una maggiore resilienza dell'applicazione. Se un servizio fallisce o presenta problemi, gli altri servizi possono continuare a funzionare senza interruzioni significative. I team di sviluppo possono quindi lavorare in modo indipendente su diversi servizi, consentendo un ciclo di sviluppo più rapido e agili processi di rilascio.

Un altro vantaggio è la facilità di adozione di tecnologie e linguaggi di programmazione diversi. Poiché ogni servizio è un'entità indipendente, è possibile utilizzare il linguaggio

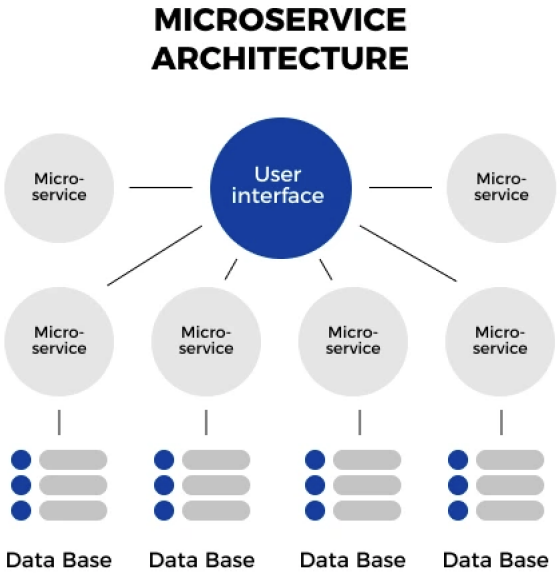

Figura 2: L'interfaccia utente è connessa a microservizi multipli, che si appoggiano ai database ove necessario

di programmazione o il framework più adatto per ogni servizio, in base alle specifiche esigenze e alle competenze del team di sviluppo.

2.4 Differenze principali tra architettura monolitica e a microservizi

Le differenze tra i due tipi di architettura presentate nelle sezioni precedenti possono essere di diversa natura, come illustrato nel prosieguo di questa sezione. Tali differenze vengono analizzate rispetto allo sviluppo, alla gestione dei database, all'aspetto di *scaling*, agli ambienti di produzione e *testing*, ai modelli di integrazione e *delivery*.

2.4.1 Sviluppo

Nell'architettura monolitica, l'applicazione è sviluppata come un'unica unità, in cui il codice sorgente, il backend e frontend sono combinati insieme. L'intero sistema viene sviluppato, testato e distribuito come un'unità entità coesa. Ciò significa che le modifiche e le nuove funzionalità richiedono l'intervento sul codice sorgente dell'intero sistema. Se anche una sola parte dell'applicazione richiede modifiche, l'intero sistema deve essere compilato e distrubuito nuovamente, rendendo il processo di sviluppo complesso e potenzialmente lento, specialmente quando il progetto cresce in dimensioni e complessità.

Con i microservizi si adotta un approccio diverso: l'applicazione viene suddivisa in un insieme di servizi più piccoli, ciascuno dei quali ha una responsabilità specifica e funziona come un modulo autonomo. Ogni servizio può essere sviluppato, testato e distribuito indipendentemente dagli altri. Questa modularità offre diversi vantaggi nello sviluppo, permettendo ai team di lavorare su diversi servizi in parallelo senza interferenze. Ogni servizio può essere scritto con linguaggi di programmazione o framework diversi, meglio adatti per la loro funzionalità e favorendo quindi una flessibilità tecnologica maggiore. I servizi possono essere scalati individualmente in base alle necessità: se un particolare

servizio richiede maggiori risorse per gestire un carico di lavoro elevato, può essere scalato in modo indipendente senza dover scalare l'intero sistema. Tra i svantaggi di questa architettura c'è la complessità aggiuntiva a causa della gestione delle comunicazioni tra i servizi, la scalabilità coordinata e la sincronizzazione dei dati che richiedono una progettazione e implementazione più attenta, portando quindi a test di monitoraggio più complessi.

2.4.2 Database

La differenza nella gestione del database tra le due architetture è sostanziale e non va sottovalutata. In un'architettura monolitica, esiste un unico database condiviso utilizzato da tutte le componenti dell'applicazione. La gestione è relativamente semplice, ma le migrazioni e gli aggiornamenti possono comportare interruzioni e richiedere attenzione.

In un'architettura a microservizi, ogni servizio ha il proprio database separato, consentendo maggiore isolamento e scalabilità. La gestione diventa più complessa, poiché possono essere utilizzati diversi tipi di database per servizi diversi. L'uso di strumenti di orchestrazione dei database è comune per gestire la distribuzione, la replica e il bilanciamento del carico.

2.4.3 Scaling

Nell'approccio monolitico, lo *scaling* è spesso limitato all'intera applicazione. Ciò significa che se una parte dell'applicazione richiede più risorse per gestire un aumento del carico di lavoro, l'intero sistema deve essere scalato, anche se altre parti dell'applicazione potrebbero non averne bisogno. Questo può portare a uno spreco di risorse. Al contrario, nell'approccio a microservizi, è possibile scalare singoli servizi in modo indipendente in base alle esigenze specifiche. Poiché ogni servizio è autonomo e può essere eseguito su diverse istanze, è possibile distribuire le risorse in modo più efficiente. Ciò consente di adattare la scalabilità in modo più preciso, ottimizzando l'utilizzo delle risorse e garantendo una maggiore scalabilità orizzontale. Di seguito, in Figura 3 e in Figura 4, si illustra come lo *scaling* orizzontale può essere applicato in entrambi gli approcci ma con risultati diversi.

Figura 3: Esempio di *scaling* orizzontale di una architettura monolitica.

2.4.4 Ambiente di produzione

Dovendo rilasciare un singolo elemento (l'applicazione), il processo di **deployment** è più semplice in un'architettura **monolitica** rispetto a quella a microservizi, che può comprendere centinaia di servizi richiedenti rilasci individuali. Come accennato precedentemente,

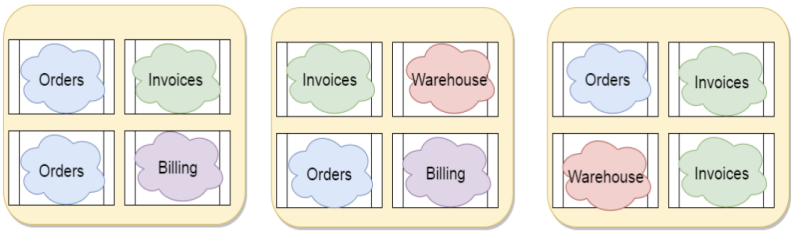

Figura 4: Esempio di *scaling* orizzontale di una architettura a microservizi.

i microservizi tendono ad essere aggiornati più frequentemente, quindi è fondamentale automatizzare l'intero processo di deployment. Inoltre, è essenziale garantire che il rilascio delle nuove versioni avvenga in modo sicuro per evitare guasti o problemi di compatibilità.

È importante che il processo di **deployment** sia controllato e tracciato per individuare eventuali problemi e correggerli tempestivamente. A tal fine, è necessario utilizzare un sistema di **gestione dei rilasci** che tenga traccia delle versioni rilasciate e dei cambiamenti apportati. Inoltre, è fondamentale che il processo di **deployment** sia **ripetibile** e **scalabile** per garantire un'alta disponibilità del servizio.

Nel caso dei microservizi, è fondamentale disporre di un sistema di orchestrazione che gestisca l'intera infrastruttura in modo automatico, coordinando i vari servizi e le risorse necessarie. Questo sistema può essere utilizzato per il **deploy** dei servizi, la **configurazione** delle risorse di rete, l'**aggiornamento** dei servizi, la gestione delle risorse di **storage** e la **scalabilità orizzontale** dei servizi.

Due esempi di sistemi di orchestrazione per microservizi sono Kubernetes e Docker Swarm. Kubernetes è un sistema open source sviluppato da Google che consente di gestire in modo efficace e scalabile un grande numero di container, offrendo funzionalità avanzate come il bilanciamento del carico, la gestione degli aggiornamenti, la scalabilità dei servizi e la riparazione automatica dei problemi. Docker Swarm è un altro sistema di orchestrazione basato su Docker, che consente di gestire un *cluster* di host Docker come se fossero un unico host.

In entrambi i casi, questi sistemi di orchestrazione consentono di gestire in modo efficace e automatico una grande quantità di servizi e risorse, semplificando notevolmente la gestione dell'infrastruttura, e riducendo i tempi di deploy e manutenzione.

2.4.5 Testing

La transizione da monolita a microservizi porta cambiamenti significativi al processo di testing. Di seguito ci concentreremo sulle differenze tra i paradigmi architetturali, discutendo delle strategie e delle *best practices*.

In un'architettura **monolitica**, il sistema è tipicamente testato come una singola entità. Questo include il testing end-to-end completo, dove l'intero *stack* dell'applicazione è validato nelle sue funzionalità e performance. Viene anche eseguito il ***unit testing*** sui componenti individuali interni al monolita.

Con i microservizi, il focus si sposta dal testare il sistema come un singolo blocco al testare servizi individuali in modo indipendente.

Le strategie di testing principali in comune sono:

- ***Integration testing***: Testing dell'interazione e integrazione tra componenti o moduli diversi. In un'architettura **monolitica**, ciò consiste nel testare l'interazione tra vari moduli o strati dell'applicazione. Nei microservizi, si focalizza nel testare l'interazione e il flusso di dati tra servizi individuali. Questi test verificano che i componenti funzionino correttamente insieme e che i dati vengano scambiati accuratamente.

- ***Unit testing***: Strategia fondamentale che consiste nel testare componenti individuali o unità isolate per verificare la loro correttezza e il loro comportamento. Gli *Unit test* vengono scritti per coprire funzioni specifiche, metodi, classi. Servono ad identificare *bugs*, gestire casi limite e assicurarsi che ciascuna unità funzioni come previsto.

- ***Functional testing***: Il testing funzionale verifica che il sistema o l'applicazione funzioni come previsto e che raggiunga specifici requisiti. Consiste nel testare il comportamento dell'applicazione contro specifiche funzionali o *user stories*. Entrambe le architetture richiedono un testing funzionale per assicurarsi che il sistema si comporti correttamente e offra le funzionalità previste.

Data la natura distribuita dei microservizi, ottenere una copertura completa di testing può essere difficoltoso. Concentrare gli sforzi di test sulle funzionalità critiche e sulle aree ad alto rischio aiuta a garantire che le parti più importanti del sistema vengano testate a fondo.

Inoltre, l'implementazione di ***pipeline* di *Continuous Integration* e *Continuous Delivery* (CI/CD)** facilita il testing automatizzato e il rilascio dei microservizi. Poiché le applicazioni **monolitiche** sono tipicamente autonome, il CI è relativamente semplice da implementare.

2.4.6 *Continuous Integration, Continuous Delivery*

L'integrazione continua (CI) in un'architettura monolitica consiste nell'integrare modifiche di codice da sviluppatori diversi in una repository condivisa.

Il CD sui monoliti si riferisce al processo di distribuire automaticamente il codice testato e validato all'ambiente di produzione. Ciò consiste nell'automatizzare il ***build***, **l'impacchettamento** e i passi di **deployment**. Dato che l'applicazione è rilasciata come una singola unità, il CD in architetture monolitiche è generalmente meno complesso in confronto ai microservizi.

Il CI in un'architettura a microservizi può essere più impegnativo a causa della natura distribuita del sistema: ciascun microservizio potrebbe avere un proprio repository e dipendenze, richiedendo coordinazione per l'***integration testing***. È importante adottare strumenti di CI che supportino il *building* e il *testing* dei singoli microservizi per facilitare l'integrazione tra di loro.

A differenza delle architetture monolitiche, il **CD** nei microservizi richiede la gestione della distribuzione di molteplici servizi con le loro proprie versioni e dipendenze e coordinare tutto ciò può risultare più complesso. Tuttavia la natura disaccoppiata dei microservizi consente una distribuzione indipendente, consentendo rilasci e scalabilità più veloci.

Differenze principali tra CI/CD in architetture monolitiche e a microservizi

1. **Difficoltà:** Implementare il CI/CD in architetture monolitiche è generalmente meno impegnativo grazie alla natura centralizzata dell'applicazione. In contrasto, le architetture a microservizi introducono complessità aggiuntive, come la gestione di repository molteplici, il coordinamento dell'*integration testing* e la gestione del deployment distribuito.

2. **Carico di lavoro:** Nel CI/CD delle architetture monolitiche, si costruisce e si testa l'intera applicazione come unità singola. Ciò può comportare tempi di build più lunghi e requisiti di risorse maggiori. Nei microservizi, il carico di lavoro è distribuito tra i singoli servizi, consentendo una parallelizzazione e cicli di *build/testing* potenzialmente più veloci.

3. **Efficienza:** I microservizi offrono il vantaggio del deployment indipendente, consentendo rilasci più veloci e frequenti. Inoltre, l'impatto delle modifiche sugli altri servizi è ridotto, poiché ciascun servizio può essere sviluppato, testato e rilasciato separatamente.

4. **Performance:** In termini di performance, i microservizi offrono una maggiore scalabilità e resilienza. L'abilità di scalare servizi singoli in modo indipendente in base alla domanda consente un utilizzo più efficiente delle risorse.

2.4.7 Sommario

L'architettura **monolitica** sviluppa l'applicazione come un'unica unità, rendendo il processo di sviluppo complesso e lento quando il progetto cresce in dimensioni. Al contrario, i microservizi suddividono l'applicazione in servizi più piccoli, consentendo lo sviluppo indipendente e parallelo, con maggior flessibilità tecnologica.

Nella gestione del **database**, l'architettura monolitica utilizza un unico database condiviso, mentre i microservizi hanno database separati per maggiore isolamento e scalabilità.

Per lo **scaling**, l'architettura monolitica richiede di scalare l'intera applicazione anche quando solo una parte ne ha bisogno, mentre i microservizi consentono di scalare singoli servizi in modo indipendente, ottimizzando l'utilizzo delle risorse.

Nell'ambiente di produzione, l'architettura monolitica semplifica il **deployment** di un'unica applicazione, mentre i microservizi richiedono un sistema di **orchestrazione** per gestire l'infrastruttura in modo automatico.

Per quanto riguarda il **testing**, l'architettura monolitica testa l'intero sistema come un'unica entità, mentre i microservizi si concentrano sul testing dei singoli servizi, richiedendo strategie di testing specifiche per l'interazione e l'integrazione tra di essi.

Infine, la ***continuous integration*** e ***continuous delivery*** sono meno complesse in un'architettura monolitica, mentre i microservizi richiedono una gestione più avanzata a causa della distribuzione di molti servizi. Tuttavia, i microservizi offrono vantaggi come il deployment indipendente e una maggiore scalabilità e resilienza.

3 Tecniche per la transizione

Effettuare una transizione da architettura monolitica ad una a microservizi richiede una pianificazione attenta e un approccio *step-by-step*. Questa pagina fornirà una panoramica delle tecniche e considerazioni coinvolte nel processo di transizione.

1. **Conoscere l'architettura monolitica corrente:**

   - Il primo passo è ottenere una conoscenza approfondita dell'architettura monolitica esistente. Identificare le componenti chiave, le dipendenze, e limiti interni al sistema. Questa analisi aiuterà a determinare quali parti del *codebase* sono adatte per la decomposizione in microservizi.

2. **Identificare gli *Enabler* per microservizi:**

   - Certe caratteristiche dell'applicazione possono comportarsi come *Enabler* per la transizione all'architettura a microservizi. Si considerano i seguenti fattori:
     a. **Modularizzazione forte**: Identificare moduli o componenti all'interno del codebase monolitico che possono essere disaccoppiati (*decoupled*) e rilasciati indipendentemente.
     b. **Chiari limiti di servizio:** Cercare limiti interni al sistema dove ha senso dividere la funzionalità in servizi separati.
     c. **Scalabilità e performance:** Identificare aree dell'applicazione che subiscono un carico elevato o richiedono specifiche misure di scalabilità. Questi possono essere buoni candidati per microservizi separati.
     d. **Domain-Driven-Design (DDD):** Applicare i principi di DDD per analizzare il modello di dominio e identificare i contesti che possono essere incapsulati in servizi individuali.

3. **Definire la strategia di scomposizione:**

   - La scomposizione è il processo di scomposizione dell'applicazione in microservizi più piccoli e distribuibili in modo più indipendente. Ci sono diversi approcci:
     a. **Scomposizione verticale:** Si inizia dividendo il monolita in base alle capacità aziendali o *feature*. Ciascun microservizio gestirà una specifica *feature* end-to-end.
     b. **Scomposizione orizzontale:** Identificare i componenti comuni o moduli che possono essere estratti in microservizi condivisi, come autenticazione, *logging* e *caching*.
     c. **Pattern Strangler:** Rimpiazzare gradualmente o aumentare le funzionalità nel monolita con microservizi. Ciò comporta il *routing* di richieste specifiche a nuovi servizi mentre il resto dell'applicazione rimane intatta.

4. **Pianificare il processo di separazione:**

   - Una volta che si ha identificato la strategia di scomposizione, bisogna pianificare il processo di separazione.

a. **Identificare le interfacce di servizio:** Determinare gli API e i meccanismi di comunicazione tra i microservizi. Ciò include definire il contratto e i formati dei dati che useranno per interagire tra di loro.
b. **Estrarre librerie condivise:** Estrarre le librerie condivise o i componenti dal monolita in moduli separati che possono essere usati dai microservizi.
c. **Stabilire la gestione dei dati:** Decidere come i dati verranno gestiti attraverso tutti i microservizi. Potrebbe essere necessario implementare l'archiviazione distribuita dei dati, *event sourcing*, o altre tecniche di sincronizzazione dei dati.
d. **Implementare l'infrastruttura:** Impostare l'infrastruttura necessaria per il rilascio e la gestione dei microservizi. Ciò include la *containerizzazione*, l'orchestrazione, la *discovery* dei servizi e il monitoraggio.

5. **Criteri per la selezione delle funzionalità:**

- Nel determinare la funzionalità per scomporre in microservizi, si considerano i seguenti criteri:
  a. **Alto valore commerciale:** Si inizia con le funzionalità che forniscono un valore aziendale significativo o hanno un impatto elevato sul sistema.
  b. **Meno parti intricate:** Si identificano i componenti nel monolita che hanno dipendenze minime o hanno un *coupling* basso con le altre parti. Questi sono più facili da estrarre come servizi *standalone*.
  c. **Nuova funzionalità:** Se si stanno introducendo nuove funzionalità o moduli, prendi in considerazione di implementarli come microservizi sin dall'inizio per mantenere una consistenza architetturale.
  d. **Conformità tecnologica e linguaggio:** Valutare se una specifica funzionalità o modulo richiede tecnologie o linguaggi di programmazione diversi che sono più adatti per i microservizi.

4 Un caso di studio nell'ambito del turismo

La transizione da un'architettura monolitica a microservizi può offrire numerosi vantaggi per le grandi applicazioni. Tuttavia, è essenziale riconoscere che questa trasformazione presenta anche diverse sfide. Per ottenere informazioni su queste sfide e sulla loro complessità, è stato condotto un caso studio nella quale verrà esaminata la transizione di una piattaforma di prenotazione crociere in una fase di progettazione e studio di fattibilità per l'implementazione di un'architettura a microservizi. Attualmente, la piattaforma opera in un'architettura monolitica consolidata, ma con l'obiettivo di migliorare l'efficienza, la scalabilità e la flessibilità del sistema, si è presa in considerazione l'adozione di un'architettura a microservizi. È importante notare che, al momento non è disponibile un *proof of concept* completo della nuova architettura, in quanto l'implementazione è ancora in fase di studio e valutazione. Tuttavia, questo caso studio fornirà un'analisi approfondita delle sfide, delle considerazioni e delle potenzialità associate a questa transizione in corso.

4.1 Introduzione

Nel presente caso studio, esamineremo questa piattaforma che utilizza *WebSphere Commerce*. *WebSphere Commerce* è una soluzione software che offre un ambiente di *runtime* per l'esecuzione e la gestione di applicazioni aziendali su una vasta gamma di piattaforme. Questo software si basa sul linguaggio di programmazione Java e offre un'architettura robusta e scalabile per lo sviluppo e la distribuzione di applicazioni *enterprise*. La sua architettura modulare consente l'implementazione di servizi e funzionalità in modo flessibile, fornendo un'infrastruttura solida per supportare applicazioni complesse e distribuite. Tuttavia, nonostante le capacità robuste, la piattaforma presenta alcune limitazioni nel contesto attuale. Ad esempio, i tempi di avvio del server in ambiente locale sono risultati lenti rispetto agli standard moderni, comportando ritardi nello sviluppo e nell'implementazione di nuove funzionalità. Inoltre, la mancanza di testing automatizzati rende difficile garantire la stabilità del software, aumentando il rischio di regressioni dopo i rilasci in produzione.

4.2 Architettura

Si è deciso di scegliere il **MACH** come l'architettura concettuale obiettivo su cui basare la transizione. L'architettura MACH, acronimo di **Microservices**, **API-first**, **Cloud-native** e **Headless**, rappresenta un approccio architetturale moderno per la progettazione di sistemi software altamente scalabili e flessibili. Questa metodologia si basa sulla suddivisione dell'applicazione in componenti modulari chiamati microservizi, i quali comunicano tra di loro attraverso API ben definite.

1. ***Microservices***: essendo piccoli e autonomi, i microservizi offrono numerosi vantaggi in quanto consentono lo sviluppo, il test e l'implementazione indipendente, fornendo un'agilità superiore nello sviluppo e nella manutenzione del software. Inoltre, l'abilità di scalare singolarmente i microservizi in base alle esigenze di carico garantisce un sistema efficiente e resiliente nel suo complesso.

2. ***Cloud-native***: questa architettura è pensata per essere *cloud-native*, il che significa che è progettata per sfruttare le caratteristiche e i servizi offerti dalle piattaforme cloud. Questo include l'uso di infrastrutture scalabili, orchestrazione dei container

e servizi di gestione del carico, che consentono di gestire il traffico in modo efficiente e di garantire l'affidabilità del sistema.

3. ***Headless***: questa caratteristica consiste nel separare la layer di presentazione dell'interfaccia utente dalla logica di backend. Ciò consente agli sviluppatori di creare esperienze front-end personalizzate utilizzando tecnologie e framework moderni, senza doversi preoccupare della complessità del backend. Inoltre, l'approccio *headless* facilita l'integrazione con diversi canali di distribuzione, come applicazioni mobili, siti web e dispositivi IoT.

Questa architettura, illustrata in Figura 5, si allinea con gli avanzamenti tecnologici attuali, consentendo all'azienda di sfruttare le nuove tecnologie e i cambiamenti nel panorama informatico. Questo permette di mantenere il sistema aggiornato e competitivo nel lungo termine, adattandosi alle esigenze emergenti del mercato.

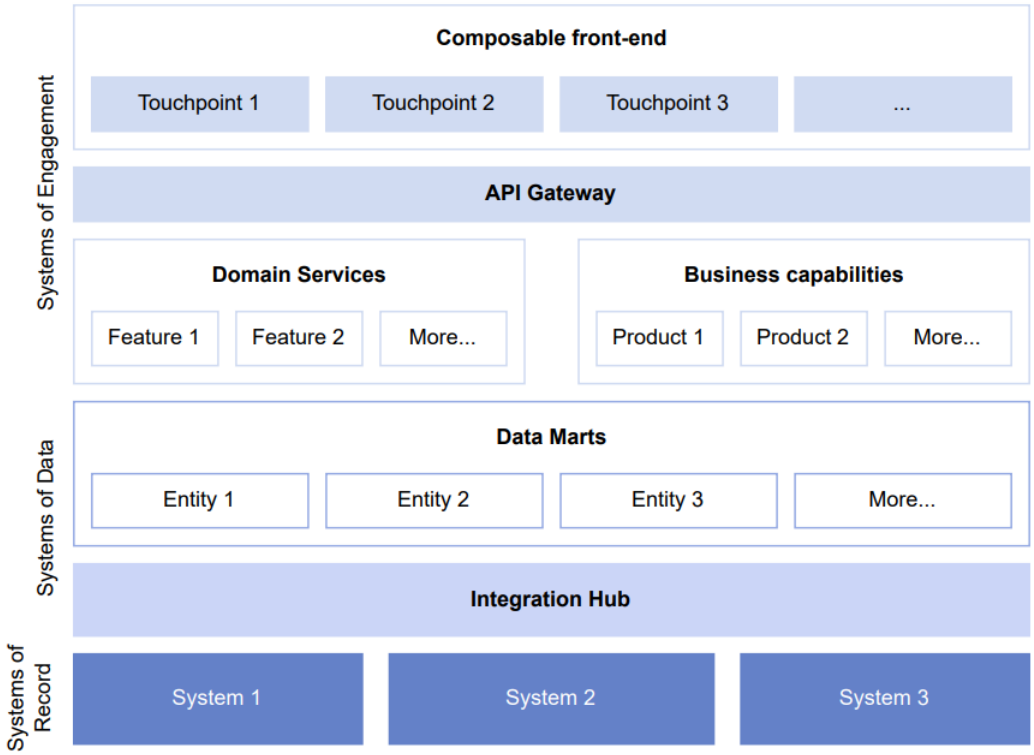

Figura 5: Design architetturale del MACH, suddiviso in tre sistemi: *engagement, data, record*

5 Approccio progettuale

L'azienda ha affrontato cinque considerazioni chiave, schematizzate in Figura 6, per identificare soluzioni ai *pain-point* più importanti. Sono state delineate le *feature* essenziali e mappate le interazioni tra i componenti. Successivamente sono state riviste tutte le integrazioni, comprese quelle con applicazioni di terze parti. Si è studiato il *lifespan* dei dati e il ciclo di vita, affrontando la sicurezza come aspetto cruciale. Infine, è stata definita una strategia CI/CD per automatizzare il rilascio del software.

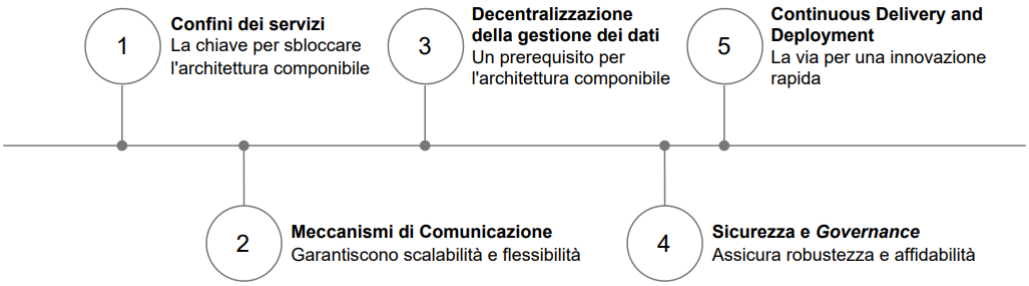

Figura 6: Le cinque considerazioni chiave.

5.1 Confini dei servizi

È stato condotto un approfondito esame dell'architettura monolitica esistente al fine di identificare i componenti separati che possono essere trasformati in microservizi indipendenti. È stato poi determinato come partizionare l'applicazione in servizi con confini (*boundaries*) e responsabilità ben definite. Ogni componente avrà il proprio spazio tecnico, completamente indipendente da altri aspetti come il linguaggio di programmazione utilizzato, le librerie, la tipologia di scalabilità e così via.

5.2 Meccanismi di comunicazione

Come presentato nella Figura 7, si è poi determinato come ciascun servizio dovrà comunicare con gli altri e interagire con i servizi di terze parti, identificando i seguenti *pattern*:

1. Comunicazione REST tramite JSON utilizzando un Registro dei Servizi (chiamato *Discovery*) tra i microservizi
2. Comunicazione REST/SOAP con terze parti
3. Importazione ETL *batch (extract, transform, load)*
4. Utilizzo di code di messaggi per il caricamento offline

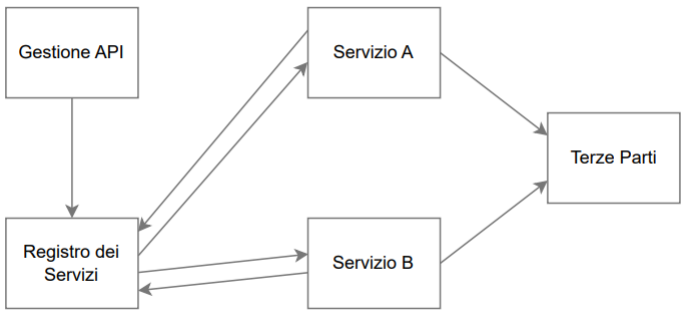

Figura 7: Meccanismi di comunicazione

È inoltre necessario riutilizzare altre componenti per sfruttare alcune logiche di business. Ad esempio, se si desidera spostare un carrello alla sezione degli ordini, sarà

necessario chiamare il servizio di *checkout* fornendo l'ID del carrello come parametro. Prima di ciò, sarà necessario validare il carrello, ma questa validazione non verrà effettuata nel servizio di *checkout*. Sarà invece chiamato il servizio del carrello per eseguire questa validazione, e poi si procederà in base alla risposta ottenuta.

5.3 Decentralizzazione della gestione dei dati

Con la transizione a microservizi, si è proposto di passare da un database centrale a più database al fine di evitare il *coupling* e fornire scalabilità, illustrato nella Figura 8. Sarà necessario comprendere come il ciclo di vita dei dati debba funzionare tra i microservizi e i servizi di terze parti, oltre all'identificare come garantire la consistenza dei dati e il corretto funzionamento delle transazione all'interno della piattaforma.

Il passaggio da un'architettura monolitica a una basata su microservizi comporta la possibilità di avere più di un database. Tuttavia, ciò può comportare il rischio di duplicazione dei dati su più database. Pertanto, i servizi di terze parti dovranno valutare con quale frequenza avvengono le modifiche al database e decidere se memorizzare i dati in un database o in un livello di *cache* per un breve periodo di tempo.

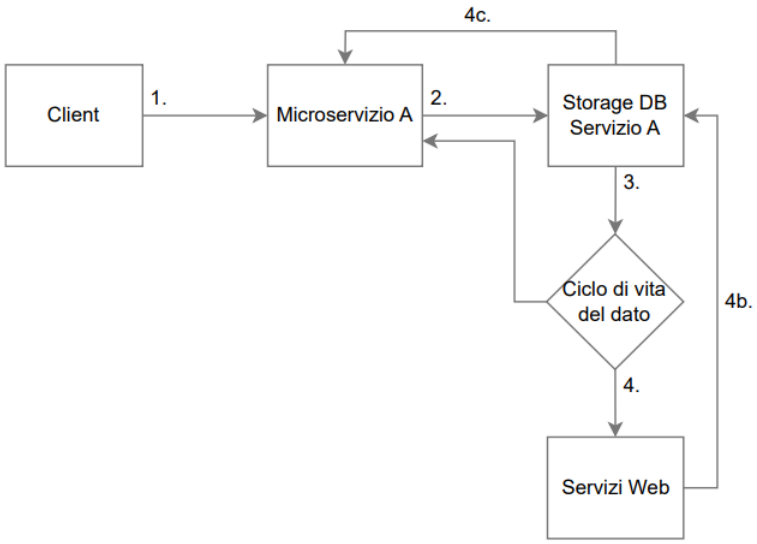

Figura 8: Grafico della decentralizzazione

5.4 Sicurezza e *governance*

È stato definito come l'architettura deve gestire le seguenti misure di sicurezza:

1. **Autenticazione e autorizzazione**
   - Token JWT, permessi, sessione WCS
2. **Crittografia e protezione dati**
   - Meccanismi di crittografia più recenti per proteggere i dati sensibili
3. **Sicurezza della rete**

(a) Segmentazione della rete

(b) Firewall

(c) (Opzionale) Sistema di rilevazione intrusioni

4. ***Logging* e *monitoring***

(a) Splunk

(b) Dynatrace

La sicurezza è uno degli argomenti più cruciali da affrontare, specialmente considerando l'implementazione di un'architettura a più strati. Come illustrato in Figura 9 stato definito un metodo di autenticazione per l'architettura, in cui sia il *WebSphere Commerce Server* (WCS) che il server con la nuova architettura saranno entrambi "accesi". Questo consentirà di passare facilmente da una piattaforma all'altra, offrendo la flessibilità necessaria per la migrazione graduale.

Per gestire l'autenticazione, verrà utilizzato un token JWT (JSON *Web Token*) in cui saranno memorizzate solo informazioni di breve durata. Ciò permette di garantire la sicurezza dei dati sensibili durante la comunicazione tra i diversi microservizi.

Inoltre, sarà necessario un sistema che gestisca in modo efficiente e senza intoppi queste misure di sicurezza. Per il monitoraggio, i migliori strumenti da utilizzare sono *Splunk* e *Dynatrace*. L'architettura che l'azienda sta progettando dovrà integrarsi facilmente con questi due strumenti, al fine di garantire una gestione efficace e una rapida risoluzione dei problemi relativi alla sicurezza.

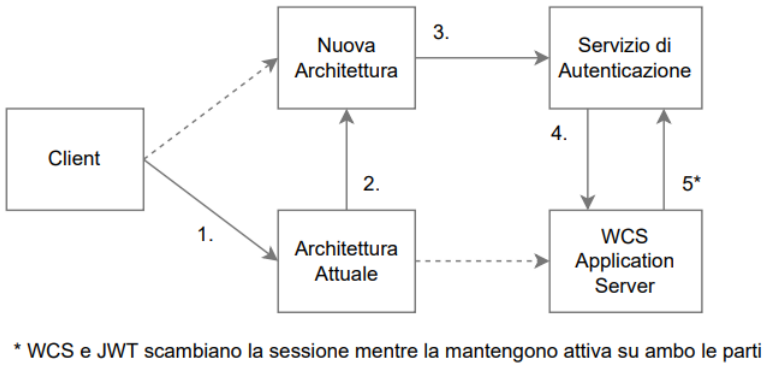

Figura 9: Sicurezza e *Governance*, si ottiene una certa flessibilità implementando due server con le due diverse architetture

5.5 *Continuous Delivery / Deployment*

Infine, si illustra la strategia del *Continuous Integration/Continuous Deployment* (CI/CD) che è stata definita. Questa strategia prevede l'automazione di vari passaggi di compilazione, tra cui l'automazione della *build*, l'esecuzione di *unit test*, *test automation* e *performance test*.

- ***Build automation***: questo passaggio coinvolge l'automazione del processo di compilazione del codice sorgente in un formato eseguibile o in un pacchetto distribuibile.

- ***Unit test execution***: gli *unit test* verificano il corretto funzionamento delle singole unità di codice (solitamente funzioni o metodi) isolate. L'automazione di questa fase comporta l'esecuzione automatica dei test unitari per identificare eventuali errori o difetti nel codice.
- ***Test automation***: questo passaggio coinvolge l'automazione dei test funzionali che verificano il comportamento del software dall'inizio alla fine. Sono eseguiti test end-to-end o test su componenti interconnesse per rilevare bug o problemi di integrazione.
- ***Performance test***: vengono misurate e valutate le prestazioni del software in termini di tempi di risposta, scalabilità, affidabilità e utilizzo delle risorse. L'automazione consente di eseguire test di carico e stress in modo automatico, fornendo dati quantitativi sulle prestazioni del sistema.

Ogni componente del sistema dovrà seguire il proprio ciclo di sviluppo e sarà sottoposto a una serie di validazioni prima di poter essere rilasciato. Ciò include l'esecuzione di *unit test*, *test automation* e *performance test*, al fine di garantire che il componente sia stabile, privo di errori e in grado di mantenere prestazioni accettabili.

Per semplificare il processo di rilascio, si è deciso di adottare la strategia *Blue-Green*. Questa strategia prevede una finestra di tempo di rilascio più breve e un processo più rapido. Sarà possibile rilasciare una nuova versione in modo asincrono e successivamente passare a essa una volta che sarà pronta. Tecnicamente, ciò viene realizzato modificando la connessione del *router* per indirizzare il traffico verso la nuova versione.

L'implementazione di una strategia di CI/CD e l'utilizzo della strategia *Blue-Green* come illustrato in Figura 10 consentiranno di accelerare il processo di rilascio, riducendo al minimo i tempi di inattività e garantendo una consegna più rapida e affidabile delle nuove funzionalità del sistema.

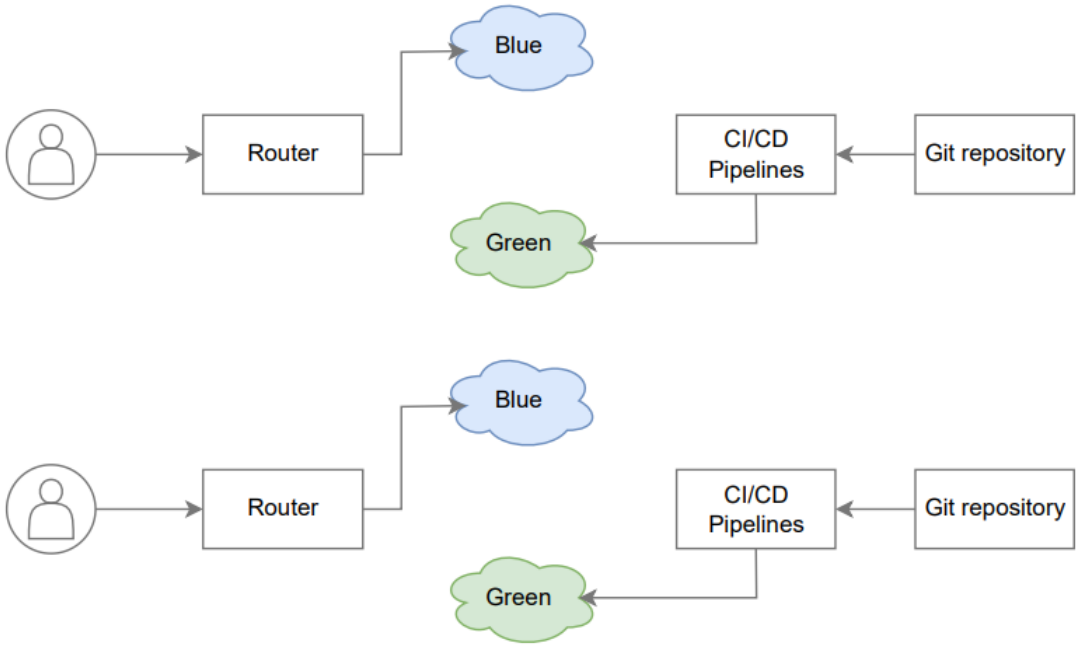

Figura 10: La strategia *Blue-Green* per migliorare il processo di rilascio

6 Soluzioni di *eCommerce*

Dopo aver delineato l'architettura, esaminato i vari aspetti come i confini (*boundaries*) dei servizi, i meccanismi di comunicazione, la decentralizzazione della gestione dei dati, la sicurezza e la governance, nonché l'approccio di *continuous delivery* e deployment, l'azienda ha il compito fondamentale di scegliere la piattaforma più adatta per queste esigenze.

Questa decisione riguarda la scelta tra l'adozione dello strumento **CommerceTools**, una soluzione di eCommerce preesistente, o l'implementazione di una piattaforma personalizzata ad-hoc.

6.1 CommerceTools

CommerceTools è una piattaforma di eCommerce di terze parti che offre un insieme completo di funzionalità per supportare le esigenze di vendita online dell'azienda. Essa si distingue per la sua natura modulare e scalabile, permettendo di sviluppare un'esperienza di acquisto personalizzata e flessibile per i suoi clienti. L'utilizzo di questa piattaforma offre una serie di vantaggi significativi, tra cui la velocità di implementazione grazie alle sue funzionalità già pronte all'uso, la gestione semplificata degli aspetti infrastrutturali e la possibilità di adattarsi rapidamente ai cambiamenti del mercato.

6.2 Vantaggi e Svantaggi

Dopo una analisi dettagliata delle caratteristiche della piattaforma, si denota che l'adozione della piattaforma CommerceTools porta i seguenti vantaggi e svantaggi:

6.2.1 Vantaggi

1. **Principi MACH**: Tutti i processi e gli strumenti sono costruiti nativamente e la piattaforma si allinea con i principi del MACH. Infatti, CommerceTools si attiene correttamente a questi principi in quanto offre un supporto nativo per i microservizi, con un approccio API-first. La comunicazione fluida tra i diversi servizi attraverso API ben definite, permette una grande flessibilità, scalabilità e sviluppo indipendente di ciascun componente. Oltre a ciò, la piattafoma è *cloud-native*, permettendo all'azienda di adattarsi velocemente alla mole di richieste, scalando efficientemente i servizi quando necessario.

2. **Caratteristiche eCommerce**: il nuovo strumento presenta un insieme di funzionalità che sono già utilizzate nella piattaforma corrente, come il carrello, lo store, l'internazionalizzazione (*i18n*), localizzazione.

3. **Struttura pronta all'uso**: il "*backbone*" è già pronto all'uso, in quanto comprende una tecnologia ben definita e una struttura di progetto consolidata. Ciò significa che la piattaforma offre una base solida e un'organizzazione chiara, semplificando l'implementazione e lo sviluppo delle funzionalità richieste, permettendo di risparmiare tempo e risorse nella fase iniziale di configurazione.

4. **Manutenibilità**: l'infrastruttura è più facile da gestire e mantenere in quanto è aggiornata regolarmente e migliorata da un team dedicato di esperti. Permettendo di minimizzare i sforzi nella manutenibilità dello strumento stesso.

5. ***Drag and drop***: è possibile personalizzare l'esperienza utente (UX) senza la necessità di apportare modifiche al codice, consentendo agli utenti di effettuare modifiche con un approccio intuitivo.

6.2.2 Svantaggi

1. **Copertura**: poiché l'industria delle crociere presenta requisiti molto particolari e processi operativi specifici, la piattaforma richiederà una ampia estensione da parte dell'azienda per adattarsi pienamente a tali concetti.

2. **API / Strumenti di importazione**: è necessario alimentare la piattaforma con dati e informazioni ottenute tramite API o strumenti di importazione specifici, che richiede una corretta integrazione dei propri sistemi aziendali o fonti di dati con CommerceTools.

3. **Limitazioni tecniche**: l'estensione dei microservizi predefiniti di CommerceTools potrebbe non essere altrettanto flessibile o scalabile come la creazione di microservizi personalizzati su una piattaforma ad-hoc, portando sfide e restrizioni nell'adattare lo strumento alle esigenze specifiche dell'azienda. Sarebbe inoltre necessario lavorare all'interno delle linee guide ed eventuali restrizioni imposte dalla piattaforma.

4. **Dipendenza dal fornitore / Costi licenze**: essendo una piattaforma proprietaria, ciò significa che, una volta adottata, l'azienda potrebbe diventare dipendente dal fornitore per la gestione e l'evoluzione della piattaforma. L'utilizzo di essa comporta anche costi legati alle eventuali licenze.

5. **Integrazione dei sistemi**: durante la transizione dal sistema corrente a CommerceTools, l'azienda dovrà mantenere entrambi i sistemi operativi per garantire un passaggio fluido e minimizzare l'impatto sulle operazioni aziendali. Questo richiederà un'attenta pianificazione e gestione delle risorse per garantire che entrambi i sistemi siano adeguatamente supportati e che l'interazione tra di essi avvenga senza problemi.

6.3 Una soluzione personalizzata

L'opzione alternativa pensata dall'azienda è una soluzione completamente personalizzata, che a differenza di CommerceTools offre un grado di flessibilità e controllo nettamente maggiore, ma ad un costo massivo in termini di risorse, sviluppo, gestione e manutenzione.

6.3.1 Vantaggi

1. **Punti critici**: La scelta di sviluppare una piattaforma personalizzata ad-hoc offre l'opportunità di affrontare specificamente tutti i problemi individuati nell'architettura attuale dell'azienda.

2. ***Full ownership***: l'azienda ha la possibilità di apportare modifiche e adattamenti all'architettura senza dipendere da fornitori esterni o restrizioni imposte da soluzioni commerciali di terze parti.

3. ***License-free***: l'adozione di una piattaforma personalizzata offre all'azienda il vantaggio di essere libera dai costi di licenza associati all'utilizzo di funzionalità limitate di una soluzione commerciale predefinita.

4. **Focus migliore**: questa scelta offre all'azienda il vantaggio di ottenere una qualità maggiore nel processo di sviluppo. Questo è possibile grazie alla possibilità per il team di sviluppo di concentrarsi completamente sulle esigenze specifiche del business.

5. **Prestazioni migliori**: essendo la piattaforma completamente personalizzata, è possibile ottimizzare l'architettura, il codice e i microservizi per garantire una performance ottimale. Si possono implementare meccanismi di caching per ridurre i tempi di risposta, oltre a eseguire test di prestazioni accurati e ottimizzare le query per ridurre il tempo di accesso ai dati.

6.3.2 Svantaggi

1. **Costi e tempistiche**: l'implementazione di una piattaforma custom comporta costi maggiori e richiede più tempo nella fase iniziale rispetto all'adozione di CommerceTools.

2. **Rischio di fallimenti, vulnerabilità**: il rischio di fallimenti nell'implementazione e vulnerabilità della sicurezza sono maggiori. Questo è dovuto alla complessità aggiuntiva associata alla progettazione e allo sviluppo su misura dei microservizi e all'integrazione dei vari componenti. Poiché la piattaforma personalizzata viene costruita da zero, sono presenti maggiori opportunità per errori di progettazione, bug nel codice e problematiche di compatibilità.

3. **Manutenzione, supporto, aggiornamenti**: sarà necessario fornire una manutenzione continua, un supporto adeguato e aggiornamenti software per garantire il corretto funzionamento e l'evoluzione del sistema nel tempo.

4. **Caratteristiche eCommerce mancanti**: a differenza di CommerceTools, sarà necessario implementare funzionalità da zero (come il carrello, lo *store*, *i18n*, localizzazione).

5. **Mancanza di documentazione**: poiché la piattaforma è creata internamente, non esiste una documentazione preesistente che possa essere consultata per comprendere le diverse caratteristiche e il loro funzionamento. Pertanto, il team di sviluppo è responsabile della creazione della documentazione, che deve essere esaustiva, chiara e facilmente comprensibile.

6.4 Conclusione

Dopo un'attenta valutazione delle considerazioni sopra esposte, l'azienda ha preso la determinazione di sviluppare una piattaforma personalizzata in virtù dei vantaggi significativi che questa scelta comporta. L'adozione di una soluzione ad-hoc offre l'opportunità di affrontare in maniera specifica tutte le criticità individuate nell'attuale architettura aziendale, consentendo un'ottimizzazione su misura per le peculiari esigenze del business. Inoltre, l'azienda si assicura la piena proprietà del sistema, avendo la facoltà di apportare

modifiche e adattamenti senza dover dipendere da fornitori esterni o subire restrizioni imposte da soluzioni commerciali predefinite di terze parti. L'eliminazione dei costi di licenza associati all'utilizzo di tali soluzioni commerciali rappresenta un vantaggio economico notevole per l'azienda. Allo stesso modo, la scelta consentirà di concentrare il focus maggiormente sulle specifiche esigenze del business, garantendo un innalzamento della qualità nel processo di sviluppo. La completa personalizzazione dell'architettura, del codice e dei microservizi consentirà di ottenere prestazioni superiori, in quanto sarà possibile implementare metodi di caching al fine di ridurre i tempi di risposta, eseguire test di prestazioni accurati e perfezionare le query per ridurre i tempi di accesso ai dati. In sintesi, il decisionale dell'azienda di sviluppare una piattaforma personalizzata rappresenta l'opzione più idonea, fornendo un insieme di vantaggi che si tradurranno in un sistema più efficiente, performante e perfettamente in linea con le particolari necessità organizzative.

7 Infrastruttura tecnologica

In questo capitolo, parleremo di quattro punti chiave che riguardano l'infrastruttura della piattaforma personalizzata ad-hoc presentata nel capitolo precedente. Questi punti evidenziano le soluzioni e gli strumenti utilizzati per garantire una gestione efficace, la sicurezza e il monitoraggio del software aziendale. Come raffigurato in Figura 11, è preparato un esaustivo schema contenente tutte le sezioni, di cui andremo a descrivere in dettaglio.

1. **Azure DNS & Front Door**: verranno adottati i servizi *cloud-based* di Azure, DNS e Front Door per gestire in modo efficiente i domini DNS e il *routing* del traffico verso le sue applicazioni web. Azure DNS offre una gestione affidabile e scalabile dei domini DNS, consentendo all'azienda di mantenere un controllo completo sui suoi indirizzi web. Front Door, d'altra parte, facilita il *routing* intelligente del traffico verso i diversi servizi, migliorando la distribuzione e l'accessibilità delle risorse online.

2. **Rete pubblica**: per le applicazioni web che richiedono accessibilità pubblica, l'azienda implementerà una rete progettata per consentire un facile collegamento alla rete internet pubblica.

3. **Rete privata**: per le applicazioni e i dati sensibili che richiedono massima sicurezza e privacy, l'azienda implementerà una rete completamente isolata. Questa infrastruttura di rete garantisce un livello elevato di protezione, impedendo l'accesso da parte del pubblico non autorizzato.

4. **Deployment e strumenti di monitoraggio**: verranno utilizzati strumenti e servizi forniti da Azure che permettono un *deploy* e gestione delle applicazioni. Sarà anche possibile monitorare le loro prestazioni e identificare eventuali problemi.

Rete pubblica (Figura 12)

- **Azure Web Apps**: L'azienda ha scelto di adottare Azure Web Apps per distribuire e ospitare sia il front-end React che il back-end GraphQL delle proprie applicazioni

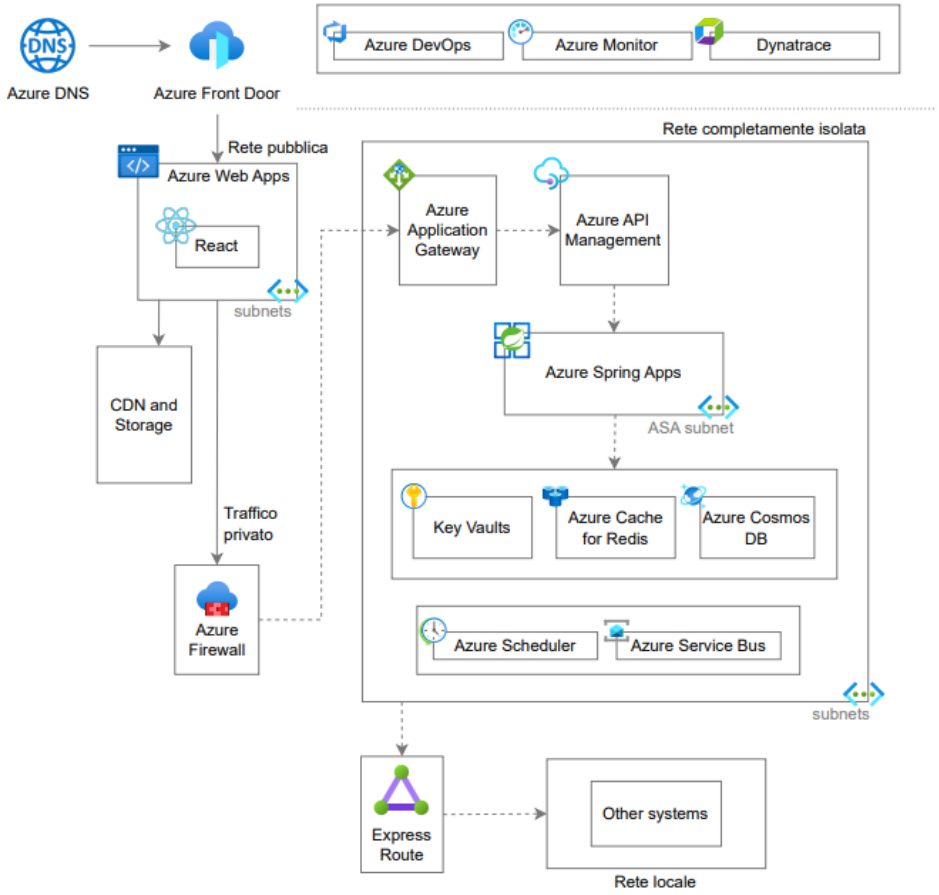

Figura 11: L'architettura completa, comprendente le varie sezioni di rete pubblica, privata (isolata), locale, strumenti di *monitoring*

web. Questa scelta consente di rendere rapidamente operative le componenti front-end e back-end, permettendo agli utenti di accedere velocemente alle funzionalità offerte dalle applicazioni.

- **Azure Firewall**: Al fine di garantire la sicurezza del back-end GraphQL, l'azienda ha optato per l'utilizzo di Azure Firewall, una soluzione avanzata per la protezione dell'infrastruttura di rete. Attraverso la configurazione del Firewall, è possibile stabilire regole di filtraggio e reindirizzamento del traffico, in modo da consentire l'accesso alle risorse back-end solo alle richieste autorizzate provenienti dal front-end. In particolare, il Firewall è stato configurato per reindirizzare tutte le chiamate da GraphQL verso una rete privata, al fine di garantire che soltanto il traffico autorizzato possa accedere alle risorse back-end. Questo approccio fornisce un livello aggiuntivo di sicurezza e protezione per l'ambiente di back-end.

Rete privata (Figura 13)

- **Azure Application Gateway**: per gestire il traffico delle applicazioni web, l'azienda ha adottato Azure Application Gateway, un servizio di bilanciamento del carico del traffico web. Questo servizio consente di distribuire in modo efficiente

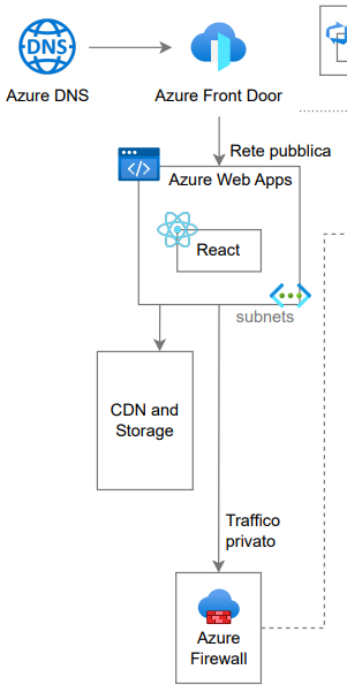

Figura 12: Rete pubblica, questa rete conterrà le tecnologie di *React* e *GraphQL*, che faranno affidamento alle CDN e alle memorie contenenti i dati

il traffico tra le sue diverse istanze. In particolare, questo servizio viene usato per distribuire il traffico tra le applicazioni basate su Node.js e i microservizi sviluppati utilizzando Spring Boot su Azure Spring Apps. Quest'ultimo è un servizio di Azure utilizzato per distribuire e scalare applicazioni Java Spring Boot.

- **Azure API Management**: per migliorare la gestione delle API e garantire la sicurezza, l'azienda ha adottato Azure API Management. Questo servizio consente di pubblicare, proteggere e gestire le API. Attraverso questo servizio, è possibile applicare politiche di sicurezza e limitazioni sui microservizi dell'azienda, gestire l'accesso alle API e monitorare le prestazioni.

- **Azure Key Vault**: per lo storage di dati sensibili, come le chiavi API, l'azienda ha optato per l'utilizzo di Azure Key Vault, un servizio basato su cloud che consente di archiviare e gestire chiavi crittografiche, certificati e altri dati segreti. È possibile integrare Key Vault con i microservizi in esecuzione su Spring Apps, garantendo un accesso sicuro ai dati sensibili senza esporli nel codice dell'applicazione.

- **Azure Cosmos DB**: per le esigenze di database, si userà Azure Cosmos DB, un servizio di database multi-modello distribuito globalmente, ideale per la creazione di applicazioni altamente scalabili. Cosmos DB supporterà i microservizi, offrendo modelli di dati NoSQL, SQL e grafo.

- **Azure ExpressRoute**: per collegare i microservizi di Spring Apps alla rete locale aziendale, l'azienda potrà fare uso di ExpressRoute, una connessione dedicata e privata tra l'infrastruttura locale e i data center di Azure. Questa connessione consentirà all'azienda di estendere l'infrastruttura di rete in Azure e accedere ai microservizi di Spring Apps in modo sicuro ed efficiente.

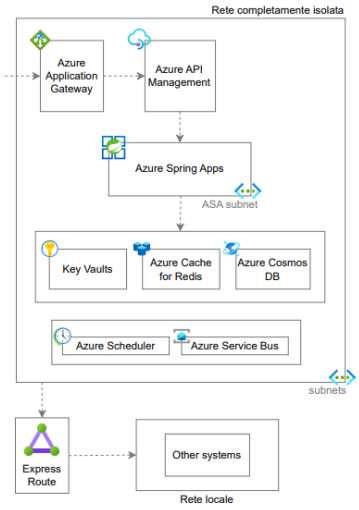

Figura 13: Rete privata, un insieme completamente isolato composto da servizi progettati per il supporto "dietro le quinte" dell'applicazione

- **Azure Cache for Redis**: per migliorare le prestazioni dei microservizi, è possibile usare Azure Cache for Redis, un servizio di memorizzazione dati in memoria basato su *Redis*, che supporta la memorizzazione di dati in formato chiave-valore, e strutture dati di documenti. Questo servizio consente di creare una *cache* dei dati più frequentemente accessati, riducendo così la necessità di richiamare ripetutamente il database.

Strumenti di Deployment e Monitoring (Figura 14)

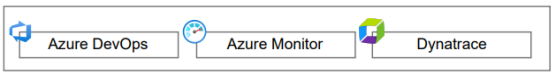

Figura 14: I tre strumenti di deployment e monitoring

- **Azure DevOps**: tra i strumenti di rilascio e monitoring, si userà Azure DevOps, un insieme completo di strumenti di sviluppo che offre una soluzione *end-to-end* per la costruzione e il rilascio di applicazioni. Questa piattaforma comprende diverse funzionalità, tra cui il controllo del codice sorgente, la creazione di pipeline di compilazione e rilascio, la gestione dei test e uno spazio dedicato alla documentazione. Supporta anche i processi agili, consentendo all'azienda di adottare metodologie di sviluppo agili come *Scrum* o *Kanban*.

- **Azure Monitor**: si utilizzerà infine Azure Monitor, un servizio di monitoraggio e analisi del cloud che fornisce visibilità all'interno delle applicazioni e delle prestazioni dell'infrastruttura. Questo servizio consente di raccogliere, analizzare e agire sui dati telemetrici provenienti dalle applicazioni e dai servizi in esecuzione. Monitor offre anche funzionalità avanzate come la rilevazione delle anomalie, l'allarme e la diagnostica.

Riepilogo

Per concludere questo capitolo, seguendo la Figura 15 evidenziamo i punti chiave dell'infrastruttura ad ora discussa; possiamo affermare che grazie alla vasta gamma di servizi offerta da Azure, l'azienda è stata in grado di sviluppare un piano completo e dettagliato che servirà come fondamenta per la futura architettura a microservizi, superando così i limiti dell'approccio monolitico ad ora adottato.

Attraverso l'implementazione di numerosi servizi, ognuno con un ruolo specifico ma altrettanto importante, l'azienda è stata in grado di affrontare efficacemente le sfide e le limitazioni riscontrate con l'architettura monolitica. Azure DNS e Front Door forniranno la gestione dei domini DNS e il *routing* del traffico verso le applicazioni web, consentendo una distribuzione più efficiente. L'utilizzo di reti esposte pubblicamente e reti completamente isolate garantirà un adeguato bilanciamento tra l'accessibilità pubblica delle applicazioni web e la sicurezza dei dati sensibili.

L'adozione di strumenti come Azure Web Apps, Azure Spring Apps e Azure API Management consentirà la distribuzione rapida e scalabile di applicazioni, microservizi e la gestione delle API, garantendo una massima efficienza e sicurezza. L'utilizzo di Azure Key Vault permetterà lo storage e la gestione sicura di dati sensibili come le chiavi API, evitando di esporli nel codice delle applicazioni.

Azure Cosmos DB, con il suo supporto per modelli di dati NoSQL, SQL e grafici, fornirà una solida soluzione di database multi-modello per immagazzinare e gestire i dati necessari per i microservizi. L'implementazione di Azure Cache for Redis migliorerà le prestazioni dei microservizi consentendo la memorizzazione nella cache dei dati frequentemente utilizzati.

Infine, l'uso di Azure DevOps semplificherà il processo di sviluppo, compilazione e rilascio delle applicazioni, mentre Azure Monitor garantirà una visibilità completa sulle prestazioni e la salute delle applicazioni, consentendo l'identificazione tempestiva di eventuali problemi.

In conclusione, grazie a questi servizi offerti da Azure e al loro ruolo complementare, l'azienda ha potuto creare una solida infrastruttura tecnologica per supportare l'architettura a microservizi.

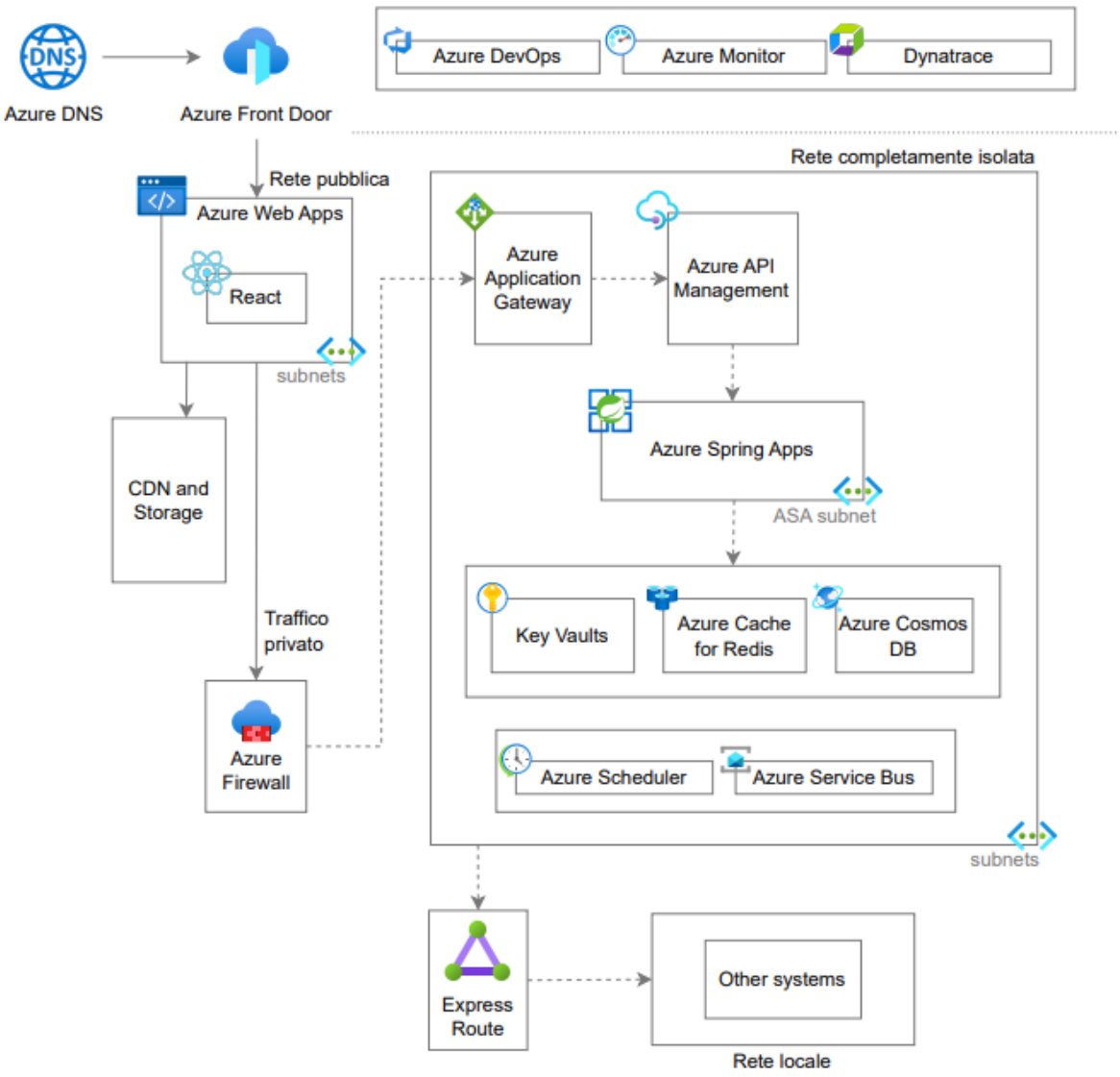

Figura 15: L'architettura ad ora vista

8 Conclusione

Durante questo percorso di ricerca, sono stati affrontati in maniera dettagliata i principali concetti e le tecniche necessarie per effettuare una transizione di successo verso l'architettura a microservizi.

Il caso studio svolto è stato di notevole importanza sia per la mia carriera nel settore, sia per il team con cui lavoro che sta pianificando tale transizione, consentendo di analizzare in dettaglio l'argomento per la sua futura implementazione. La collaborazione con i colleghi aziendali ha arricchito ulteriormente l'analisi, portando a una comprensione approfondita delle sfide e delle opportunità associate alla transizione verso la nuova architettura.

In particolare, la sezione sull'approccio progettuale ha offerto una guida chiara e pratica per la definizione dei limiti dei servizi, dei meccanismi di comunicazione, e della gestione dei dati in ingresso e in uscita, ponendo enfasi sulla sicurezza e la *governance*, garantendo che la nuova architettura rispetti i requisiti di sicurezza e consenta una gestione efficace dei servizi.

Dopo aver preparato tutti gli elementi chiave della ricerca, sia dal punto di vista teorico che pratico, è stato possibile selezionare con attenzione due soluzioni possibili, ognuna con i suoi vantaggi e le sue peculiarità. Alla fine, la scelta è ricaduta sulla piattaforma completamente personalizzata, per cui si è reso necessario un approfondimento sull'infrastruttura tecnologica, che ha portato alla decisione di adottare la suite Azure, dimostrata in grado di soddisfare la maggior parte delle esigenze discusse nel corso della ricerca. Grazie alla vasta gamma di servizi offerti da Azure, l'azienda è stata in grado di sviluppare una solida infrastruttura tecnologica che supporta l'architettura a microservizi e fornisce le basi per futuri sviluppi e scalabilità.

Concludere questa tesi rappresenta non solo un importante traguardo accademico, ma anche il culmine di un intenso percorso di studio durato tre anni, anni in cui ho avuto l'opportunità di studiare e approfondire concetti teorici e competenze pratiche per comprendere e analizzare un argomento quanto complesso quanto rilevante tutt'ora nel mondo informatico. La stesura di questa tesi è stata un'esperienza arricchente e stimolante, in quanto mi ha permesso di applicare conoscenze acquisite durante la mia carriera universitaria e lavorativa, iniziata grazie ad una passione per l'informatica presente sin dall'infanzia, messa in pratica già dagli anni di scuola superiore, dai banchi di scuola di un istituto tecnico informatico.

Il confronto tra la teoria e la pratica è stato particolarmente gratificante, poiché mi ha permesso di comprendere appieno l'importanza di considerare diversi aspetti, come la progettazione, la comunicazione tra i servizi, la gestione dei dati e la sicurezza. Inizialmente nel percorso, mi era difficile immaginare quanto fosse complesso tale settore, ma grazie all'università ho avuto l'opportunità di entrare in profondità in tantissimi argomenti complicati e altrettanto interessanti, come la progettazione di algoritmi, le reti di sistemi, l'intelligenza artificiale. Ogni corso e i progetti che includevano mi hanno permesso di sviluppare ampiamente le competenze che mi sono servite per entrare nel settore del lavoro come *software engineer*, oltre a rendermi altamente preparato per il settore tecnologico che è da tempo una delle rivoluzioni più importanti dell'era moderna.

Felice della conclusione del mio percorso triennale, sono entusiasta di proseguire gli studi nel campo informatico per continuare ad ampliare le mie conoscenze e abilità.

Riferimenti bibliografici

[1] Kevin Goetsch (2017) *Microservices for Modern Commerce*, O'Reilly Media.

[2] Eberhard Wolff (2016) *Microservices: Flexbile Software Architecture*, Addison-Wesley Professional.

[3] Frank P. Moley III (2018) *Microservices Foundations*, LinkedIn Learning.

[4] Daniel Khan (2021) *Software Architecture: Breaking a Monolith into Microservices*, LinkedIn Learning.

[5] Merkel D. (2014) *Docker: lightweight linux containers for consistent development and deployment.*

[6] Khatereh Yamini (2019) *Moving from local server to cloud service using Microsoft Azure*, Politecnico di Torino.

[7] Henrik Alderborn (2022) *The use of Microsoft Azure for high performance cloud computing - A case study*, Uppsala Universitet.

[8] Ramesh Ghimire (2020) *Deploying Software in the Cloud with CI/CD Pipelines*, Haaga-Helia University of Applied Sciences.

[9] Wikipedia (2023) *HCL Commerce*. Disponibile online: `https://en.wikipedia.org/wiki/HCL_Commerce`.

Diritti d'autore