POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Conversione di un'applicazione monolitica in un'applicazione a microservizi e confronto prestazionale tra le due architetture

Relatore
prof. Enrico Macii

Candidato
Giosuè Valfrè

Relatore Aziendale
Guido Bonino

Anno Accademico 2021-2022

Ringraziamenti

Un ringraziamento particolare ai miei genitori, per aver sempre creduto in me e avermi spronato ad andare avanti quando si sono presentate delle difficoltà lungo il mio percorso.

Grazie a Greta e a tutta la mia famiglia, ai parenti che ci sono sempre stati e a quelli che si sono aggiunti negli anni.

Grazie a Lorenzo e a tutti gli amici che mi sono stati vicini durante questo percorso.

Grazie al mio relatore per la disponibilità che ha sempre dimostrato nei miei confronti e l'aiuto che è stato pronto a darmi

Infine un grazie speciale alla TecnoLogic s.r.l. per avermi accompagnato durante questo lavoro e ai miei colleghi che sono sempre stati pronti ad aiutarmi e motivarmi in qualunque momento.

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione generale

[22, 25] Venne pronunciato per la prima volta nel 2011, alla fiera di Hannover, il termine Industria 4.0. In esso è racchiuso un significato molto importante perché con queste parole si fa riferimento alla quarta rivoluzione industriale. Nel mondo occidentale vengono ricordate tre grandi rivoluzioni industriali, a partire dalla nascita della macchina a vapore nel 1784, proseguendo con la produzione di massa grazie all'uso dell'energia elettrica nel 1870, e finendo nel 1970 con la nascita dell'informatica. Tuttavia, è ai giorni d'oggi che è in corso la quarta rivoluzione industriale: prodotti e processi interconnessi per mezzo delle nuove tecnologie e dell'IoT.

In questa nuova era è diventato di fondamentale importanza tracciare ogni processo e conservare una grande quantità di dati, riguardanti la produzione, i ritardi e le problematiche riscontrate in una catena produttiva. E', inoltre, richiesto di analizzare questa grande mole di dati in maniera semplice e veloce, per risolvere qualunque tipo di difficoltà nel minor tempo possibile incrementando i profitti.

In questo contesto e per far fronte a queste necessità, negli ultimi anni, sono nate numerose applicazioni in grado di immagazzinare grandi quantità di dati, elaborarli e renderli disponibili con semplicità a chi ne ha bisogno. Da un'indagine condotta dall'Osservatorio IoT 2022 del Politecnico di Milano si evince come, prendendo in considerazione 95 grandi imprese, l'80% di esse abbia attivato servizi a valore aggiunto in ambito Industrial Internet of Things.

Dopo aver maturato qualche anno di esperienza, interfacciandomi con grandi aziende del settore Automotive, nello sviluppo di soluzioni web IoT per Industria 4.0, ho potuto osservare i limiti di alcune decisioni prese durante la costruzione di un nuovo prodotto. Con questo progetto di tesi, intendo migliorare un'applicazione cloud attualmente in uso nel settore Automotive, trasformando l'attuale architettura monolitica in una più moderna architettura a microservizi. Nel fare questo,

voglio analizzare i vantaggi che si possono ottenere da questo tipo di design, sia nelle prestazioni che nella manutenzione dei vari servizi.

Nelle pagine successive inizierò descrivendo le principali caratteristiche di un'applicazione con architettura monolitica, continuando nei capitoli successivi con le varie metodologie di sviluppo che più si avvicinano a uno stile di programmazione moderna. In seguito racconterò le peculiarità di un'architettura a microservizi. Verrà descritta l'applicazione che si intende migliorare, proseguendo con un approfondimento sulla nuova moderna applicazione. Infine, negli ultimi capitoli, si effettueranno diversi test prestazionali, arrivando alle conclusioni sul lavoro svolto.

Capitolo 2

Architettura Monolitica

[27, 28] La parola “monolitco” ha origini antiche e il suo significato rapprensenta il fulcro dell’architettura monolitica: formato da un solo blocco. Questa tipologia di struttura nello sviluppo software è stata per molti anni la più utilizzata perché in linea con le necessità del momento.

2.1 Caratteristiche

Un software sviluppato usando un’architettura monolita è costituito da una serie di componenti fortemente dipendenti gli uni dagli altri, e proprio per questa ragione non può essere facilmente suddiviso per poter riutilizzare una parte delle sue funzionalità. Questa composizione necessita, in caso di modifiche marginali ad una singola funzionalità, un aggiornamento completo dell’applicazione e non del singolo componente interessato.

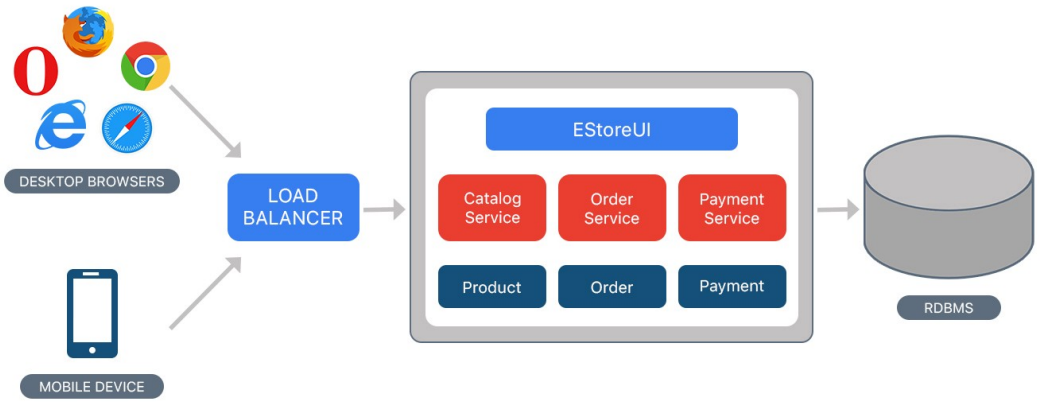

Figura 2.1. Architettura monolitica

2.2 Vantaggi

Essendo stato un modello di riferimento per molti anni, presenta ovviamente dei vantaggi da non trascurare e che tutt'oggi potrebbero essere decisivi nella scelta di una metodologia di sviluppo.

- Il design di questa architettura consente uno sviluppo piuttosto rapido se rapportato ad altre metodologie. Questo perché tutto ciò di cui necessita il software è presente all'interno della stessa applicazione e viene quindi meno tutta la complessità legata alla comunicazione tra servizi diversi

- Riducendo al minimo la comunicazione tra i vari componenti interni all'applicazione si ha un vantaggio anche da un punto di vista prestazionale, con un primo avvio più rapido

- Durante ogni passaggio di dati tra un servizio e l'altro ci sono dei rischi legati alla sicurezza dei dati che transitano. Questi rischi possiamo dire che sono notevolmente minori in un'applicazione monolitica per via di una ridotta supercie d'attacco e per l'uso di un database centralizzato

- Un ulteriore vantaggio è dato dal debugging. Essendo ogni componente dell'applicazione strettamente collegato, nel caso di modifiche, un malfunzionamento è probabile che si manifesti in maniera evidente. Inoltre, essendo in genere applicazioni di dimensioni più contenute, le fasi di test sono più rapide

2.3 Svantaggi

Trattandosi di un design datato, purtroppo, si presentano dei notevoli limiti quando si parla di manutenzione, flessibilità e aggiornabilità.

- Quando si ha un'applicazione abbastanza grande e con tutti i componenti insieme si possono avere dei seri problemi a capire dove effettuare modifiche al codice e a farlo rapidamente. Aumenta, anche, il livello di difficoltà nel comprendere a cosa effettivamente serva una data porzione di codice
- La scarsa flessibilità deriva dal limite di sviluppare l'intera applicazione con un unico linguaggio di programmazione. Il mondo dello sviluppo software vanta moltissimi linguaggi di programmazione, ognuno con caratteristiche adatte a diverse soluzioni
- Uno degli svantaggi più grande è, tuttavia, l'aggiornabilità. Per effettuare un semplice aggiornamento, anche nel caso della risoluzione di un bug poco impattante, è necessario aggiornare l'intera applicazione con il rilascio di una nuova versione. Per questa motivazione, l'architettura monolitica, non è adatta a una moderna logica di distribuzione continua CI/CD

Capitolo 3

CI/CD e DevOps

3.1 CI/CD

[1] Continuous integration (CI) è un approccio per lo sviluppo software che suggerisce di scrivere piccole porzioni di codice, testarle (in modo tale da garantire l'affidabilità del software) e caricarle in un repository condiviso da altri sviluppatori. Così facendo tutto il team è sempre aggiornato con gli ultimi cambiamenti.

Unitamente alla parte di integrazione continua, questo approccio propone Continuous Delivery / Deployment (CD), ovvero l'automazione delle fasi successive. Quando il codice ha superato una serie di test, spesso automatizzati, è pronto per essere caricato in produzione.

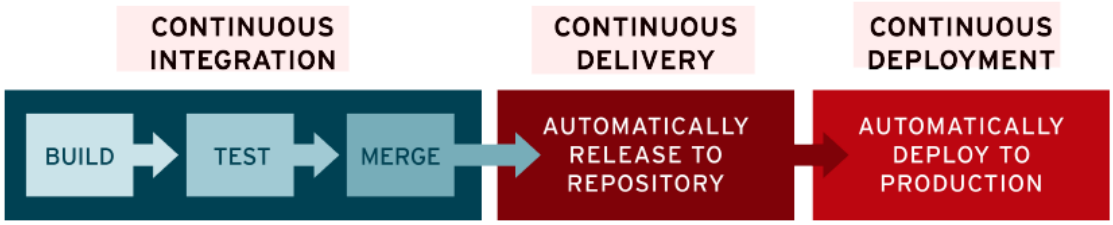

Figura 3.1. Flusso continuous integration e continuous delivery

Con questo approccio si utilizza un alto livello di automazione che consente di avere una nuova versione del software a pochi minuti dalla fine dello sviluppo. Queste caratteristiche si sposano con alcuni principi della metodologia Agile:

- Alto livello di automazione, per concentrarsi solamente sulla programmazione e non sulle attività secondarie

- Rilascio di software rapido e continuo per soddisfare il cliente

L'integrazione e il deployment continui (CI/CD) sono anche alla base della moderna metodologia di sviluppo DevOps.

3.2 DevOps

[2, 3, 4] Con DevOps si intende sviluppo (Dev) e operazioni (Ops). I team dedicati allo sviluppo e quelli dedicati alla produzione vengono uniti o fatti collaborare a stretto contatto in modo tale da avere un'unica unità in grado di seguire l'intero ciclo di vita dell'applicazione.

Da questo approccio se ne traggono numerosi vantaggi:

- Velocità, processi più snelli permettono di seguire al meglio i cambiamenti di mercato e far fronte a nuove esigenze

- Distibuzione rapida e affidabilità, grazie all'approccio CI/CD, descritto in precedenza

- Scalabilità, utilizzando dei metodi messi a disposizione è possible gestire infrastruttura e processi su qualsiasi scala, ad esempio utlizzando le infrastrutture come codice

- Collaborazione, facendo lavorare insieme i team di sviluppo e produzione si può senz'altro migliorare la collaborazione. Nasce un'automatica esigenza di condividere informazioni e comunicare utilizzando, talvolta, anche applicazioni adatte allo scopo

- Sicurezza, la velocità con cui i rilasci di nuove versioni avvengono non deve far trascurare questo aspetto importante. E' per questo si è pensato di utilizzare delle polocy di sicurezza automatizzate, che consentono così di applicare un approccio DevOps senza tralasciare l'aspetto della sicurezza. Quando un'infrastruttura è descritta attraverso il codice può, infatti, venire monitorata e convalidata tramite automatismi

Per ottenere tutti i benefici possibili ci vengono in aiuto diversi tool DevOps, ognuno dei quali è pensato per automatizzare una diversa fase del processo.

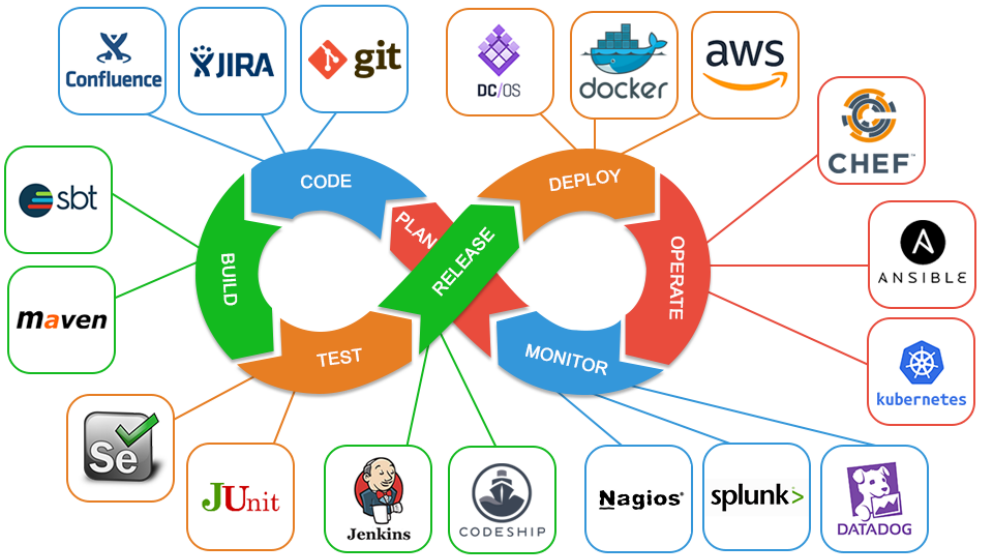

Figura 3.2. Tool utili per l'automatizzazione delle varie fasi di sviluppo

La metodologia DevOps non consente solamente di accellerare la creazione delle vecchie applicazioni monolitiche, ma vuole creare delle applicazioni strutturalmente diverse. Applicazioni che siano pensate appositamente per far fronte alla necessità di una distribuzione continua di software. Le applicazioni che meglio rispondo a questa esigenza presentano una struttura a microservizi collegati tra di loro attraverso delle API.

Capitolo 4

Architettura a Microservizi

4.1 Caratteristiche

[26, 10] L'architettura a microservizi è uno stile di sviluppo delle applicazioni che consente di dividere grandi applicazioni in componenti più piccoli, ognuno dei quali deve essere in grado di funzionare e comunicare in modo indipendente. Ogni microservizio è progettato per far fronte a una determinata necessità e l'unione di tutti questi servizi porta a un'applicazione più efficiente.

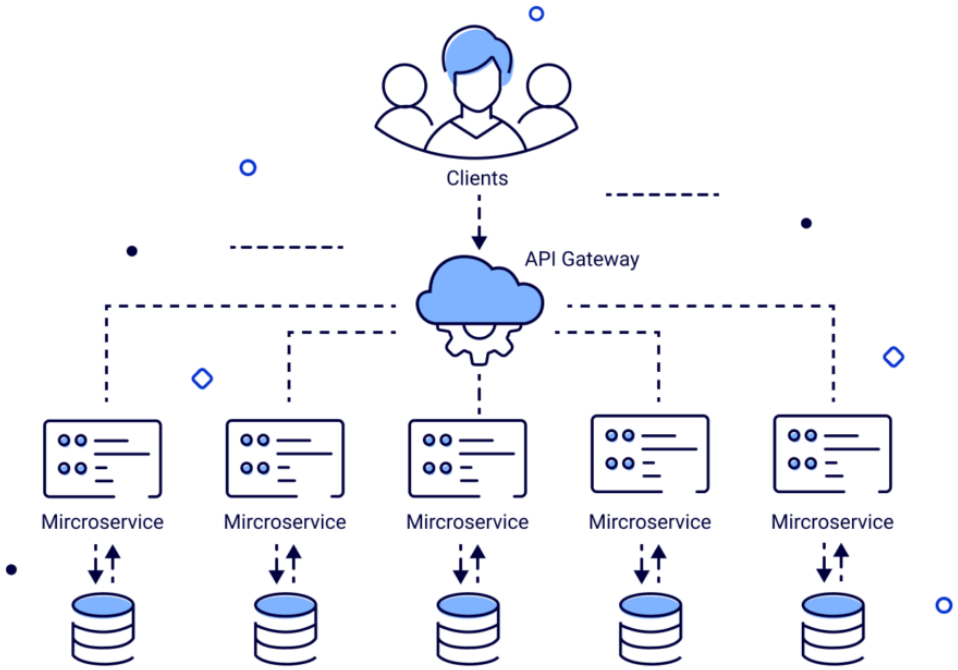

Figura 4.1. Architettura a Microservizi

Utilizzando un pattern di questo tipo, un utente può generare una richiesta tramite l'interfaccia grafica, che viene suddivisa tramite un gateway API a uno o più microservizi. Il risultato di questo meccanismo è che problemi complessi possono essere risolti in maniera relativamente semplice tramite la combinazione di diversi microservizi.

4.2 Vantaggi

I principali punti di forza di un'architettura a microservizi sono:

- Aumento della produttività; la suddivisione in frammenti piccoli e meno complessi di un problema più grande, come suggerisce il paradigma *divide et impera*, semplifica la creazione e la manutenzione. Inoltre, così facendo, si possono utilizzare per le varie parti linguaggi di programmazione differenti, diverse tecnologie e ambienti software svariati. Questa suddivisione permette anche di poter lavorare contemporaneamente tra team di sviluppo diversi

- Migliore stabilità; in caso di guasto su un singolo microservizio, considerando che siano sviluppati in maniera indipendete gli uni dagli altri, c'è l'isolamento del problema che non si ripercuote sull'intera applicazione. Anche i tempi necessari al recupero di un guasto sono inferiori perché si deve intervenire su un singolo modulo
- Maggiore scalabilità; a causa della natura indipendente dei microservizi è facile riconoscere quali sono i più critici e scalarli corretemente per aumentare le prestazioni, volendo anche distribuendo quei servizi su più server
- CI/CD; questo approccio, già descritto in precedenza, si sposa perfettamente con un pattern a microservizi potendo, dunque, sfruttare benefici come l'automatizzazione di processi manuali e la condivisione del carico di lavoro. Tutto questo si traduce in un ridotto ciclo di vita del processo di sviluppo

4.3 Svantaggi

Questa architettura presenta anche alcuni svantaggi:

- La complessità di un sistema distribuito; oltre alla gestione sicuramente più complicata anche la fase di test in un sistema distruibito diventa complessa
- La comunicazione tra i vari servizi; pur essendo indipendenti tra di loro spesso i vari microservizi devono comunicare tra di loro e questo comporta il rischio di dei fallimenti durante la fase di comunicazione
- La gestione di molti servizi e del load balancing tra di essi

Capitolo 5

Applicazione Monolitica

5.1 Descrizione

Prima di entrare nel dettaglio circa la struttura dell'applicazione che si intende trasformare con un'architettura a microservizi vanno fatte alcune premesse riguardo agli obiettivi che l'applicazione stessa deve raggiungere.

L'ambito di utilizzo è in aziende del settore automotive di dimensioni notevoli che vogliono monitorare tutto ciò che avviene nelle loro catene produttive, guardando i dati con granularità differenti: lo stabilimento, un singolo dipartimento all'interno dello stabilimento, una singola linea o addirittura la singola stazione.

Ogni stazione utilizza uno o più PLC per gestire tutto ciò che riguarda la sicurezza (sensori e barriere), le movimentazioni o le varie operazioni effettuabili dai componenti che cosituiscono la stazione stessa. Il singolo PLC può essere prodotto da diverse aziende e in base alla casa produttrice viene programmato con linguaggi di programmazione differenti. La sua programmazione è molto vicina a un linguaggio macchina, dato che vengono gestiti i singoli bit e costrutti logici come AND ed OR, ma spesso effettuata con programmi avanzati che permettono una programmazione grafica.

Lo scopo dell'applicazione, presa in analisi, è quello di raccogliere quante più informazioni possibili dai vari PLC, elaborare questi dati in maniera efficiente e renderli fruibili da remoto tramite una web application.

Le varie parti che compongono una stazione, o una linea, devono comunicare tra di loro attraverso una rete locale che per motivi di sicurezza non deve essere accessibile dall'esterno e sarebbe preferibile anche non consetire l'accesso ad internet. Per risolvere questo problema si può creare un ambiente cloud non accessibile dall'esterno e non connesso alla rete internet, ad esempio creando una macchina virtuale

all'interno della rete aziendale e creare uno spazio dedicato alla web application e all'archiviazione dei dati.

Entrando nello specifico dell'applicazione scelta è composta da due software differenti che sono in stretta comunicazione tra di loro e vengono installati in ambiente UNIX Debian.

Il primo software, che chiameremo **supervisor**, si occupa della comunicazione col PLC e di una prima elaborazione dei dati. Altro ruolo fondamente è l'invio di questi dati, già scremati per evitare il trasporto di elementi inutili, alla seconda applicazione, che chiameremo **cloud platform**.

Il supervisor comprende sia un server che un'inerfaccia frontend alla medesima porta. Comunica con i vari PLC utilizzando degli appositi helper, differenti in base alla tipologia del PLC, che gestiscono la connessione e la lettura in polling dei vari blocchi di dati presenti nel controllore. Questa fase richiede una considerevole configurazione per indicare all'applicazione cosa effettivamente ci occorre leggere, come parsificare i dati in lettura e dopo quanti millisecondi bisogna ricercare cambiamenti nelle strutture configurate. Ci sono diversi motori di elaborazione che si occupano di ricevere dei segnali in input elaborare i dati ricevuti, con specifiche funzioni, e inviare il tutto secondo un formato standard al cloud platform. La comunicazione tra supervisor e cloud platform avviene tramite connessione socket e la gestione dei messaggi per quando la connessione è assente utilizza una coda che storicizza su database i messaggi da inviare al ritorno della connessione.

Il cloud platform quando viene avviato gestisce sia la parte server che di interfaccia utente come fosse tutto un unico grande servizio. Ha grande importanza il lato frontend perché rappresenta l'interfaccia che gli utenti dell'azienda devono utilizzare per la lettura dei dati che interessano, per le varie analisi di qualità e di produzione. Tutti i segnali che raggiungono il cloud platform vengono gestiti tramite un'apposita pagina che li suddivide già in base al supervisor di provenienza. Da questa sezione è possibile configurare ogni segnale assegnando un nome, una descrizione, la stazione al quale si riferisce e persino se si vuole storicizzare sul cloud o utilizzare solamente come segnale realtime. Le pagine di visualizzazione dati sono diverse. Una è dedicata interamente ai segnali ricevuti in tempo reale e viene utilizzata per monitare in maniera veloce ed efficace quello che avviene nello stabilimento anche da chi si trova seduto nel proprio ufficio, magari distante qualche chilometro. Una pagina è dedicata a tutto ciò che viene storicizzato ed è quindi possibile analizzare dati di più mesi e raggrupparli in base alle proprie esigenze, visualizzando i grafici richiesti dall'azienda per le proprie analisi. Altro aspetto fondamentale è quello della tracciabilità dei singoli pezzi che vengono prodotti. Ipotizziamo ad esempio che una ventola montata su un determinato modello

di macchina abbia creato diversi problemi. In questo caso può tornare sicuramente utile avere una sezione dedicata all'analisi di tutti i parametri di tracciabilità che ci hanno fornito le stazioni che si sono occupate delle lavorazioni di quella determinata ventola. Ecco perché è presente anche una pagina dedicata a questo tipo di analisi. Per l'elaborazione realtime dei dati sul cloud platform vengono utilizzati degli elaboration engine che effettuano calcoli specifici in base al dato da calcolare.

Per fare un esempio concreto possiamo parlare di un indicatore di qualità molto importante nel settore dell'industria 4.0: OEE. L'OEE (Overall Equipment Effettiveness) è un indicatore percentuale che comprende tutte le tipologie di inefficienza legate alla macchina presa in considerazione. Questo indicatore è definito dal prodotto di 3 componenti: Availability, Quality, Performance.

- Availability = Tempo in cui la macchina è stata disponibile / Tempo in cui la macchina avrebbe dovuto produrre
- Quality = Pezzi prodotti buoni / (Pezzi prodotti buoni + Pezzi prodotti scartati)
- Performance = Pezzi buoni * Tempo ciclo teorico della macchina / Tempo in cui la macchina è stata disponibile * Tempo ciclo teorico della macchina

In questo caso un elaboration engine riceve in input i dati necessari ad effettuare questi calcoli e restituisce un valore finito per ognuna delle tre componenti e uno generale per l'OEE, non dovendo così delegare questi calcoli al frontend che deve mostrare questi valori e graficarli.

Adesso entriamo nel dettaglio delle tecnologie utilizzate dal supervisor e dal cloud platform per svolgere tutte le operazioni sopra descritte.

5.2 Tecnologie utilizzate

5.2.1 Node.js

[13, 23] Node.js è un ambiente di esecuzione che da' la possibilità di eseguire codice scritto in Javascript, un linguaggio nato esclusivamente per il web lato client, come un qualsiasi altro linguaggio di programmazione. Da quando è nato nel 2009, dal motore runtime Javascript di Google, V8, è stata una svolta colossale perché ha iniziato a permettere lo sviluppo anche lato server di codice Javascript, uno dei linguaggi più conosciuti al mondo. Node.js oltre ad essere open-source e cross-platform possiede caratteristiche che lo rendono ideale per lo sviluppo di una web application. Le applicazioni girano su un unico processo e non creano nuovi thread

con l'arrivo di altre richieste. Inoltre, le funzioni native e quelle delle varie librerie sono principalmente asincrone, ovvero quando viene chiamata una funzione che richiede operazioni di I/O il codice non si blocca aspettando che essa finisca, ma prosegue e il risultato viene restituito una volta pronto permettendo di risparmiare molti cicli di attesa alla CPU.

Le versioni di Node.js si dividono in tre tipologie:

- Current, dalla data di uscita per i successivi sei mesi
- Active LTS (Long-term support), scaduti i sei mesi le verisioni current diventano active. Questo, però, avviene solamente per le versioni pari, mentre quelle dispari non vengono più aggiornate. Il supporto sulle versioni Active LTS si estende fino a 30 mesi e questo ne fa preferire l'utilizzo soprattutto in produzione, dove si ha la necessità di avere un prodotto affidabile.
- Maintenance, è lo stato in cui una versione pari non è più Active ma continua ad essere aggiornata per i 30 mesi successivi

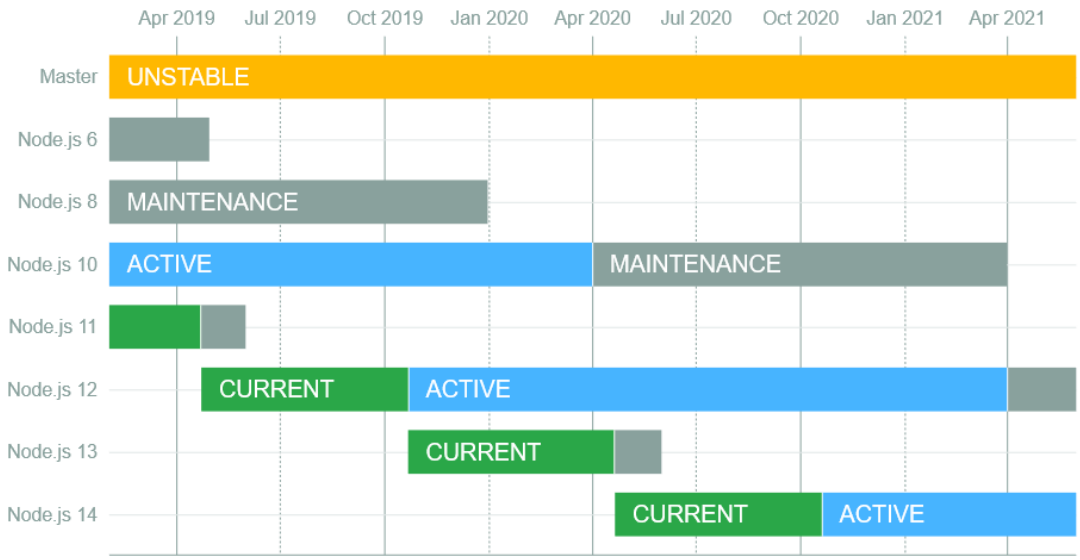

Figura 5.1. Diagramma delle versioni Node.js

L'applicazione scelta viene installata utilizzando solamente versione pari di Node.js, e in particolare dalla versione 14 in avanti.

5.2.2 Express.js

[6] Express.js è un framework server utilizzato con Node.js che fornisce funzionalità molto utili come la creazione di robuste API di routing, la possiblità di impostare dei middleware per rispondere alle chiamate http e l'integrazione con motori di templating (ad esempio EJS).

Di seguito una breve porzione di codice che mostra come sia semplice creare API utilizzando express:

```
1 var express = require('express');
2 var router = express.Router();
3
4 router.get('/', function (req, res) {
5 console.log(`Esempio su Express!`);
6 res.send(200);
7 });
```

5.2.3 Sequelize

[15] Per la comunicazione con i database utilizziamo sia sul supervisor che sul cloud platform, un ORM che funziona con Node.js chiamato Sequelize. Questo ci fornisce un'interfacciamento standard con i vari tipi di database supportati: Postgres, MySQL, MariaDB, SQLite, Microsoft SQLServer, Amazon Redshift e Snowflake's Data Cloud.

Per le nostre applicazioni utilizziamo database Postgres. Ecco un esempio con la funzione findAll, per richiedere la lista degli utenti presenti nella tabella "users" che abbiano il nome "Mario":

```
1 let usersList = await db.users.findAll({
2   where: { name: `Mario`}
3 });
```

Sequelize garantisce il supporto a transazioni, relazioni tra tabelle e vari tipi di carimento dati.

5.2.4 Redis

[14] Redis è un archivio di dati in memoria, open-source e con formato chiave valore. Date le sue eccellenti performance nei tempi di risposta è largamente utilizzato come cache e nelle soluzioni IoT.

Sul supervisor e sul cloud platform viene utilizzato per mantere dei dati in cache e per recuperarli in maniera facile e veloce successivamente. Vengono inoltre utilizzati, in alcune situazioni, i messaggi pub/sub che permettono comunicazioni realtime senza il salvataggio persistente in memoria e che vengono ricevuti solo da chi si è registrato tramite subscribe al messaggio corretto.

5.2.5 Docker

[5] Docker è una piattaforma che aiuta la gestione delle proprie applicazioni. Il suo ruolo è quello di separare l'applicazione dalla propria infrastruttura, permettendo lo sviluppo di software veloce che possa girare nell'ambiente più adatto alle proprie esigenze. Questo ambiente, isolato e sicuro, viene chiamato container.

L'utilizzo dei container docker si sposa perfettamente con il concetto di continuous integration e continuous delivery (CI/CD). Per condividere il proprio lavoro con i colleghi è sufficiente condividere un container, si può testare creando un'immagine della propria applicazione che giri in un ambiente di test e una volta effettuato lo step del debug si può pushare l'immagine aggiornata sul proprio repository e aggiornare comodamente le vecchie immagini in produzione.

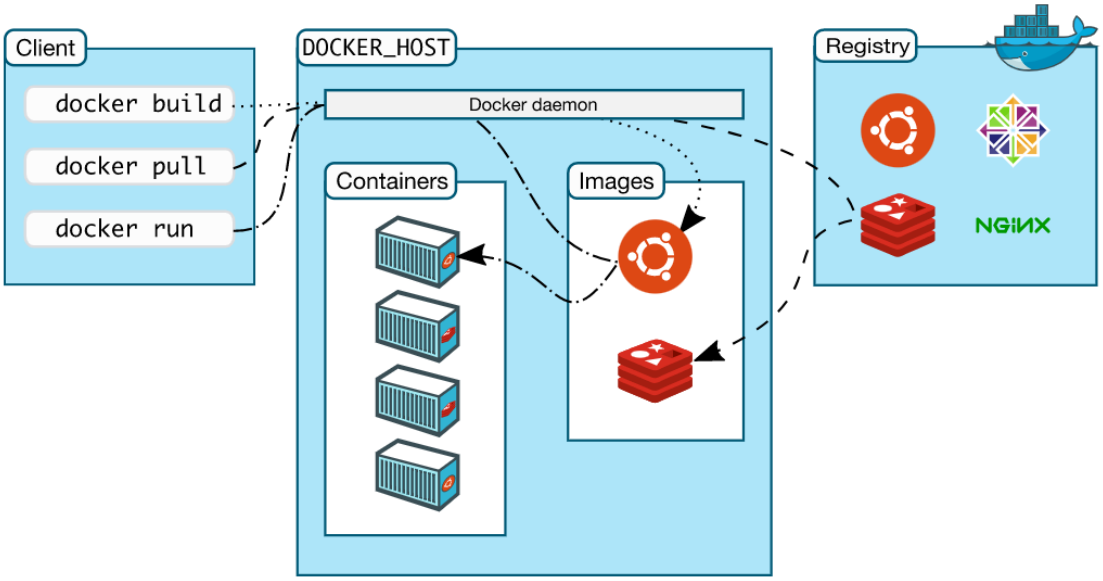

Figura 5.2. Architettura di Docker

I containers possono essere completamente isolati o collegati tra di loro, qualora debbano comunicare, attraverso delle reti interne. Esiste anche un modo per gestire

il run di una serie di containers collegati tra di loro in modo semplice e veloce, senza dover agire sul singolo elemento. In questo caso, si parla di docker-compose. Un file semplice da configurare in cui vengono indicati i vari container da creare, i vari file con variabili di environment da utilizzare, le porte che si vogliono mappare sull'host di esecuzione e molte altre cose. Una volta configurato questo file, con estensione *.yml, basterà lanciare da terminale il comando:

```
1 docker-compose up -d
```

In un primo step si è cercato di suddividere la vecchia applicazione monolitica in più servizi, per cercare almeno di rendere il deployment e i successivi aggiornamenti più facili da eseguire.

Per il supervisor si è pensato ad un servizio in grado di generare backend e frontend di configurazione dell'applicazione, un servizio per il database postgres, un servizio per redis e un ultimo servizio opzionale per PgAdimn (una web interface di visualizzazione del database).

Figura 5.3. Servizi docker attivi per il supervisor

Per il cloud platform in maniera del tutto analoga si è pensato ad un container contenente backend e frontend, uno per il database postgres, uno per redis e uno per PgAdmin. Ovviamente, nel caso in cui supervisor e cloud platform girini sullo stesso host si può utilizzare un solo servizio per PgAdmin, mentre rimangono completamenti separati i vari database.

Figura 5.4. Servizi docker attivi per il cloud platform

Nonostante questa struttura abbia permesso di velocizzare installazioni e aggiornamenti presentava ancora diversi limiti d'utilizzo, basti pensare al caso in cui il backend è sovraccarico di lavoro e per questo motivo ne risente anche il frontend. Sfruttando i microservizi sarà possibile avere più istanze di ogni servizio che tramite il load balancing non dovrebbero raggiungere livelli di stress.

5.2.6 Socket.IO

[16] Socket.IO è una libreria molto comoda e di cui facciamo gran uso sia sul supervisor che sul cloud platform. Viene utilizzata per la comunicatione bidirezionale e basata su eventi tra client e server.

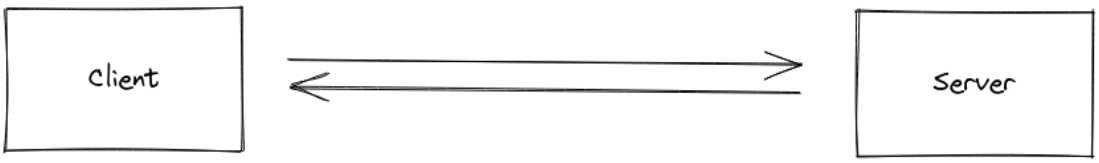

Figura 5.5. Comunicazione bidirezionale Socket.IO

Tuttavia, l'uso più importante che ne viene fatto è quello dell'invio dei dati dal supervisor al cloud platform.

5.2.7 i18n

[7] i18n è una libreria che viene utilizzata per le traduzioni. Vengono gestite diverse lingue tramite file JSON in una cartella locale. Ogni file corrisponde a una lingua

diversa e al loro interno si possono inserire le traduzioni che si desiderano.

Lato frontend il motore di template utilizzato, nel nostro caso EJS non farà altro che cercare nel giusto file JSON la traduzione corrispondente alla parola.

Di seguito un esempio, usando la sintassi EJS:

```
1 <%= __('Hello') %>
```

5.2.8 Knockout

[9] Per il lato frontend, viene utilizzata la libreria JavaSript Knockout che aiuta nelle situazioni in cui è necessaria un'interfaccia grafica che si aggiorni dinamicamente. Utilizza il design pattern MVVM (Model – View – ViewModel).

La libreria mette a disposizione dei bindings standard che raggiungono quasi tutte le esigenze. Ad esempio, il binding "foreach" permette di scorrere un array e fare determinate operazioni per ogni suo elemento, mentre il binding "text" consente di visualizzare del testo che potrebbe cambiare dinamicamente nel corso del tempo.

Capitolo 6

Applicazione a Microservizi

6.1 Descrizione

Nella nuova piattaforma non c'è una netta distinzione come tra supervisor e cloud platform. I vari compiti sono distinti dai diversi microservizi. Il software è separato in due progetti differenti: uno per il backend e un altro per il frontend.

Quando viene avviato il server viene creto in automatico un broker moleculer che si occuperà della creazione e gestione di tutti i servizi. I servizi da avviare vengono passati attraverso una variabile d'ambiente.

Si è deciso di dockerizzare anche questa applicazione per facilitarne l'installazione, rendendola così indipendete dal sistema operativo, e per agevolare i futuri aggiornamenti.

Ci sarà un container per il frontend e un container per ogni nodo backend che si desidera istanziare. Come per la vecchia applicazione sono necessari i container per i vari database utilizzati.

SQLite è il database utilizzato per salvare le configurazioni dei vari PLC da interrogare. InfluxDB viene utilizzato per il salvataggio di tutte le metriche ricavate direttamente da PLC o elaborate da appositi microservizi, e per il salvataggio dei log di sistema. PostgreSQL è, invece, sfruttato per la memorizzazione di tutti i dati restanti. Ogni database può essere istanziato più di una volta in base alle necessità.

Sono diversi i servizi che sono stati sviluppati e ognuno di essi è in grado di creare diversi microservizi per gestire i propri compiti. I vari servizi istanziabili dall'applicazione sono:

- deviceManager
- entityManager
- gateway
- gro
- logger
- metricManager
- templateManager
- webApi

6.2 Tecnologie utilizzate

Alcune delle tecnologie utilizzate sono le stesse della vecchia applicazione e sono già state raccontate nelle pagine precedenti.Vengono di seguito citate per completezza:

- Node.js
- Express.js
- Sequelize
- Redis
- Docker
- Socket.IO
- i18n

Tuttavia, per lo sviluppo della nuova piattaforma sono state utilizzate funzionalità che hanno richiesto l'uso di nuove tecnologie, che andremo a descrivere nelle pagine che seguono.

6.2.1 Moleculer

[11, 18] Moleculer è un framework per Node.js che permette la creazione di servizi affidabili e scalabili. Considerato uno dei 20 migliori framework per Node.js del 2022, è veloce e potente nell'ambito dei microservizi. Per dimostrare la notevole velocità è il sito ufficiale di moleculer a mettere in primo piano due grafici rappresentanti rispettivamente il numero di richieste al secondo effettuate localmente e quelle remote utilizzando un transporter.

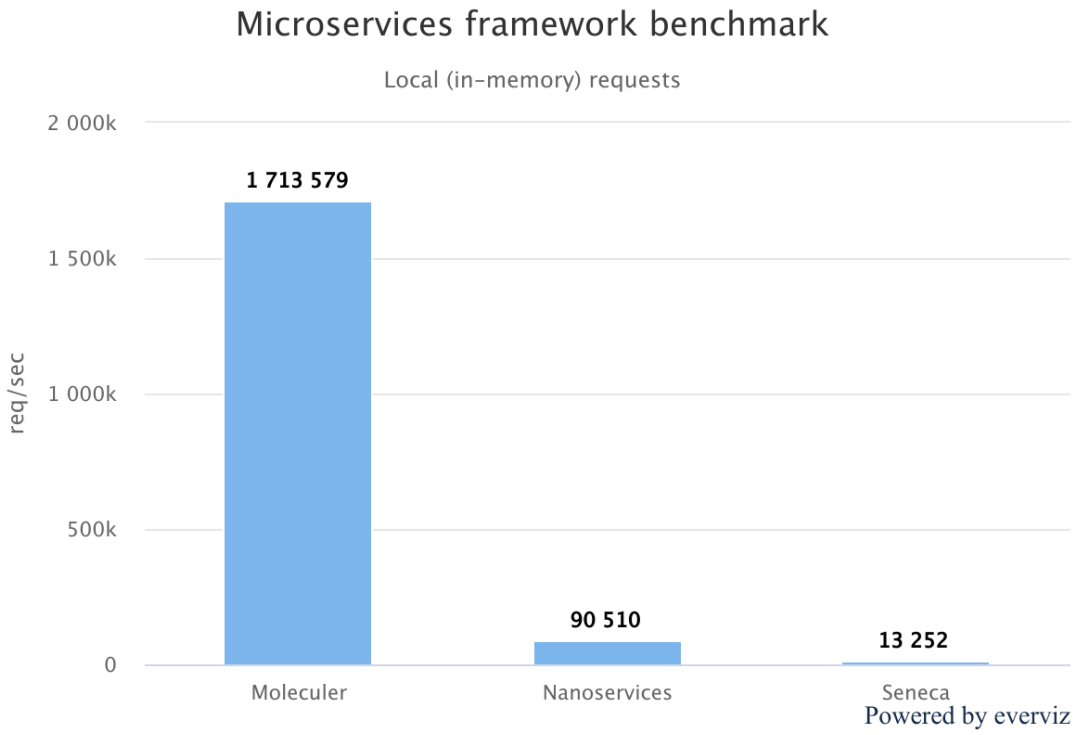

Figura 6.1. Richieste locali

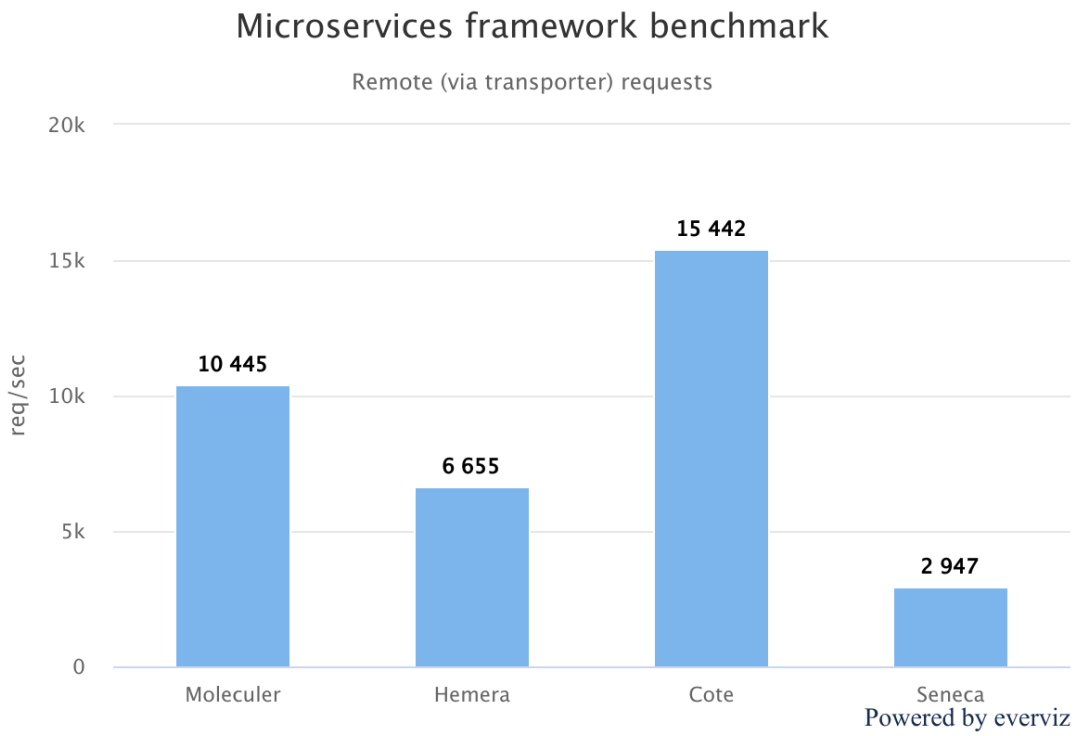

Figura 6.2. Richieste remote

Oltre all'incredibile velocità, questo framework offre numerose funzionalità per costruire e gestire i microservizi:

- Sintassi basata sulle promise, ma compatibile con sintassi await/async
- Capacità di fare load bancing delle richieste sulla base di diversi fattori, come ad esempio uso della CPU e latenza
- Tolleranza e gestione degli errori
- Supporto degli Stream
- Mixin, per evitare la duplicazione del codice
- Integrazione con transporter
- Integrazione con logger
- Integrazione con serializzatore

- Integrazione con un validatore di parametri
- Soluzione di cache dei dati già integrate
- Un API gateway

Avviare un ecosistema gestito da Moleculer non è difficile. Vediamo, utilizzando il codice dell'applicazione, come si può creare il broker capace di gestire, avviare e stoppare i vari microservizi.

```
1  const { ServiceBroker } = require('moleculer');
2
3  new ServiceBroker({
4    nodeID: process.env.FULL_NODE_ID,
5    logger: true,
6    transporter: {
7      type: 'NATS',
8      options: {
9        url: config?.nats?.url,
10       token: config?.nats?.token,
11     }
12   },
13   validator: {
14     type: 'Fastest',
15     options: {
16       // Allow to use custom methods for data validation
            useNewCustomCheckerFunction: true,
17       // Message to throw when validation failed
18       messages: {
19         uniqueConstraint: 'A record called {actual} is already
              saved',
20         missingDependency: 'Cannot set the field because {actual}
              is missing',
21         invalidInformation: 'The {field} is invalid', once: 'The {
              field} cannot be set more than once',
22       }
23     }
24   }
25 });
```

6.2.2 Nats

[12] Nats è un'infrastruttura che consente lo scambio di dati attraverso dei messaggi. I servizi Nats sono gestiti da uno o più server che vengono adeguatamente configurati per permettere il collegamento dell'intera infrastruttura.

La gestione dei messaggi è semplice ed intuitiva. Un messaggio viene invato da un *Publisher* ed è identificato da una stringa. I servizi che desiderano ricevere determinati messaggi non fanno altro che restare in ascolto in qualità di *Subscriber* e una volta ricevuto, il messaggio, viene decodificato e processato.

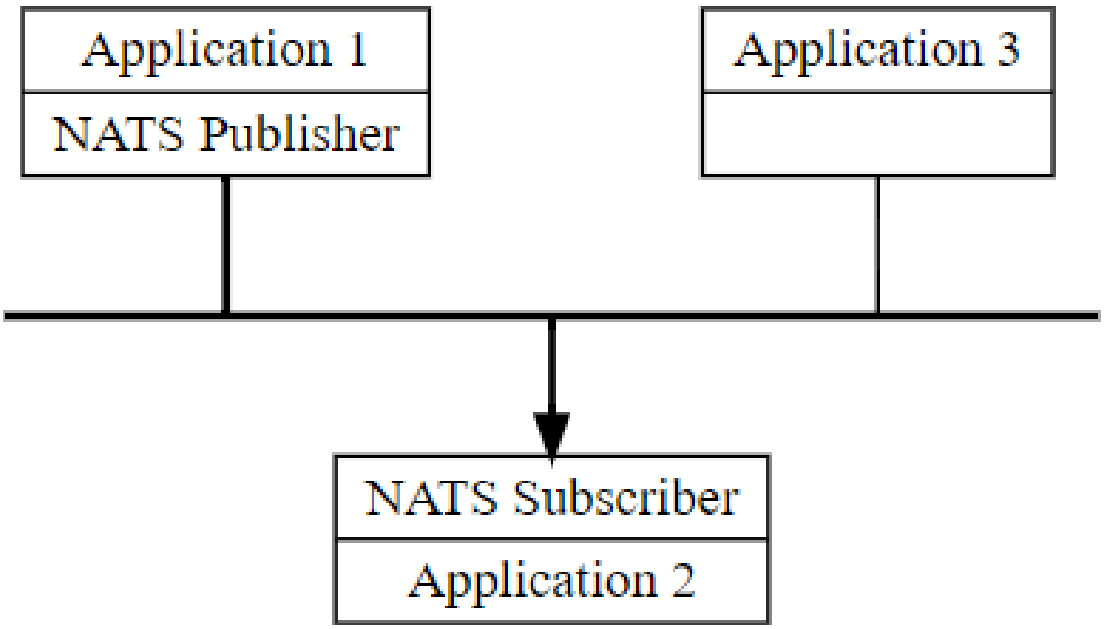

Figura 6.3. Design della struttura di messaggi NATS

Proprio per queste sue funzionalità Nats viene utilizzato nell'applicazione come transporter Moleculer.

Quando un servizio Moleculer chiama una action, è presente, di default, un load balancer interno che decide se inoltrare la richiesta localmente o se inviarla verso altri nodi. Stabilito il destinatario il messaggio viene pubblicato tramite Nats, e gli altri nodi in ascolto si occuperanno di recuperarlo e inoltrarlo ai singoli servizi. Nats permette, inoltre, la comunicazione di più nodi senza che questi conoscano i relativi indirizzi IP.

6.2.3 Influx

[17, 24] Influx è stato sviluppato dall'azienda InfluxData e viene utilizzato come sistema di gestione di database. Con InfluxDB 2.0 si utilizza il nuovo linguaggio

di programmazione Flux, ottimizzato per il processo di Estrazione, Trasformazione e Caricamento nei database (ETL). Per descrivere il funzionamento di questo database va fatta una breve digressione sui database time-series.

Un database time-series è specializzato nell'archiviazione e interrogazione di dati che hanno una dimensione misurata come unità di tempo. Un esempio può essere la misurazione effettuata da un sensore che deve storicizzare un valore ogni secondo. Questo tipo di database è caratterizzato da una mole di dati considerevoli, che, tuttavia, si riescono a interrogare velocemente.

Viste le caratteristiche dell'applicazione abbiamo deciso di utilizzare Influx per storicizzare tutti i dati che vengono letti regolarmente dal PLC, ed eventualemente quelli che hanno subito una elaborazione e devono essere memorizzati. Inoltre, lo utilizziamo per la memorizzazione di tutti i log di cui vogliamo tenere una traccia. Prendiamo come esempio il microservizio *influx_logger* creato appositamente per gestire i log dell'applicazione attraverso Influx.

La creazione del microservizio avviene utlizzando la funzione, che mette a disposizione il broker interno a Moleculer, *createService(service)*

```
1 const service = require(
2   join(__dirname, 'm_services','influx.js' )
3   );
4
5 await broker.createService(service);
```

Ora entriamo all'interno del file *influx.js* per vedere com'è strutturato questo servizio.

```
1  const influxDB = require('@mixins/influxDB.js');
2
3  module.exports = {
4
5    name: 'influx_logger',
6    mixins: [influxDB],
7
8    settings: {
9      bucketName: 'Log',
10     measurement: formats['log'],
11   },
12
13   actions: {
14   /**
15    * Log into influx
16    *
```

```
17      * @param {object} ctx molecular's context  of the call
18      * @returns {boolean} true if the data was insert,
19      *  false otherwise
20      */
21     log: {
22       async handler(ctx){
23         /*Parse data in order to send it into influxDB mixin*/
24         const point = this.dataParser(ctx.params);
25         const result = await this.push(point.measurement, point.data
               );
26         return result;
27       }
28     },
29
30
31     /**
32     * Get a page of logs
33     *
34     * @param {string} start start time to query influx
35     * @param {string} end end time to query influx
36     * @param {object} filters column for filtering data
37     * {column_name: value, column_name:value, ...}
38     * @param {number} page number of the page (for data pagination)
39     * @param {number} pageSize number of elements in a page (for
        data pagination)
40     */
41     getLogPage:{
42       rest: 'GET /log_page',
43       params: {
44         //...this.settings.timeParams,
45         start: { type: 'string', pattern: regex.ISOFormat, custom (
              start, errors, schema, name, parent, ctx) {
46           const { end } = ctx.data;
47           // Check if 'start' is before 'end' using moment
48           if (start && end && moment(start).isSameOrAfter(end))
49             errors.push({ type: 'dateMax', field: 'start', expected:
                   end });
50           return start;
51         } },
52         end: { type: 'string',  pattern: regex.ISOFormat },
53         filters: { type: 'object', optional: true, custom (filters,
              errors, schema, name, parent, ctx) {
54           /* Checks if object properties are the same as defined in
                 the measurement */
55           if(ctx?.meta?.service){
56
57           /* Get measurement prop list */
58           const names = ctx.meta.service.getAllNames();
59           for(const prop in filters){
```

```
60            if(names.indexOf(prop)===-1)
61              errors.push({type: 'objectStrict', field: 'filters',
                    actual: prop});
62          }
63          }
64          return filters;
65        } },
66        page: { type: 'number', min: 1 ,optional: true, default: 1,
              convert: true },
67        pageSize : { type: 'number', min: 1 ,optional: true, default
              : 10,convert: true },
68      },
69      async handler(ctx){
70
71      /* Parse filter */
72      const filters = {};
73      for(const filter in ctx.params.filters)
74        filters[filter] = { value: ctx.params.filters[filter],
              math_op: '==',logic_con: 'and'};
75      const { start, end, page, pageSize } = ctx.params;
76
77      /*Get total number of logs and the requested page*/
78      const [total, tuples] = await Promise.all([
79        this.getTuplesNumber('log', start, end, filters),
80        this.getPortionOfTuples('log', start, end, filters, pageSize
              ,(page-1)*pageSize)
81      ]);
82      /* Take only necessary data */
83      const preview = this.tablePreview(tuples);
84      return { total, page, pageSize, rows: preview };
85      }
86    },
87
88    },
89
90    methods: {
91    /**
92    * Format raw logger data into writable influx data
93    *
94    * @param {object} data raw logger data
95    * @returns {object} writable influx data
96    */
97    dataParser(data){
98    const point = {};
99    const item = {};
100   /* Parse data from log */
101   /*Data that have to be provided (timestamp+tags)*/
102   [
103   'level', 'service', 'node', // Default
```

```
104   'message', 'reqID', // Custom
105   'duration', 'url', 'response', 'userID' // HTTP
106   ].forEach(key => {
107     if(data[key])
108       item[key] = data[key];
109   });
110   /* Add timestamp */
111   item['time'] = data.timestamp;
112
113   /* Error logs */
114   if(data?.stackTrace){
115     item['stack'] = data.stackTrace;
116   }
117
118   point['measurement'] = 'log';
119   point['data'] = item;
120   return point;
121   }
122   }
123 };
```

Come abbiamo potuto notare dal codice vengono utilizzati dei mixin, una delle caratteristiche offerte da Molecular di cui abbiamo parlato nelle pagine precedenti, e ospitano alcune delle funzioni che leggiamo all'interno del microservizio. Le azioni che consente questo servizio sono quella di *log*, per storicizzare un log utlizzando Influx, e quella di *getLogPage*, per ottenere una lista paginata dei log memorizzati utilizzando anche alcuni parametri per filtrare i dati.

6.2.4 Vue - Nuxt

[20] Vue.js è un framework Javascript utilizzato per la creazione di interfacce utente. Viene utilizzato nell'applicazione per il frontend ed ha un pattern MVVM, similarmente a quanto visto con Knockout. Permette di creare componenti e modelli, che per evitare ripetizioni inutili di codice, possono essere riutilizzati nelle varie pagine frontend.

[19] Si è deciso di associare a Vue un altro framework, chiamato Nuxt, che comprende una grande raccolta di librerie e componenti ufficiali di Vue, in grado di semplificare lo sviluppo tra backend e frontend. Nuxt può, inoltre, essere usato per la creazione di applicazioni web renderizzate lato server.

Capitolo 7

Test Prestazionali

7.1 JMeter

[8] Apache JMeter è un'applicazione open source scritta completamente in Java. E' stata disegnata per poter effetturare dei test di carico sulle Web Application e tenere traccia dei risultati. Dalla sua configurazione permette la creazione di diversi gruppi di thread ed è impostabile un rate medio per stabilire quando attivare ogni singolo thread. Questo permette di analizzare le performance di un'applicazione sotto differenti livelli di carico.

7.2 NGINX

[21] Nginx è un web server utilizzato da grandi aziende, come Netflix, anche come bilanciatore di carico. Oltre ad essere open source, si comporta in maniera particolarmente efficiente nelle situazioni con molte richieste simultanee. Per ogni richiesta web utilizza un approccio asincrono basato sugli eventi, e questo gli consente di processare più richieste contemporaneamente senza bloccare le altre.

7.3 Test API

7.3.1 Ambiente di test

Per l'analisi ho effettuato dei test chiamando le API delle due applicazioni e registrando i risultati ottenuti. Ogni test effettuato ha una durata di 60 secondi e un numero variabile di richieste al secondo effettuate (req/sec).

Durante lo svolgimento degli esperimenti ho notato un limite nelle capacità del servizio *Web_Api* di redistribuire la grande mole di richieste arrivate ai vari microservizi attivi. Per questo motivo ho deciso di utilizzare NGINX, in modo tale da poter attivare più *Web_Api* e gestire il bilanciamento del carico tra le *Web_Api* istanziate.

La configurazione utilizzata con NGINX è la seguente:

```
events { worker_connections 10000; }

http {

    # List of application servers
    upstream api_servers {
        server cloud_next:3999;
        server cloud_next-2:3999;
        server cloud_next-3:3999;
        server cloud_next-4:3999;
        server cloud_next-5:3999;
    }

    # Configuration for the server
    server {

        # Running port
        listen [::]:5100;
        listen 5100;

        # Proxying the connections
        location / {
            proxy_pass         http://api_servers;
        }
    }
}
```

Per lo svolgimento dei test di carico sulle due applicazioni ho utilizzato un notebook con le seguenti caratteristiche:

- Processore Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz 2.11 GHz
- RAM installata 32,0 GB
- Sistema Operativo Windows 11 Pro

Nelle pagine che seguono, vedremo i risultati di numerosi test e per identificare in maniera rapida l'applicazione oggetto dell'analisi chiameremo:

- **Cloud Platform** la vecchia applicazione monolitica
- **Cloud Next 1** la nuova applicazione a microservizi (con un solo microservizio attivo)
- **Cloud Next 5** la nuova applicazione a microservizi (con 5 microservizi attivi)
- **Cloud Next 5 NGINX** la nuova applicazione associata all'utilizzo di NGINX (con 5 microservizi attivi)

7.3.2 Singola richiesta

Di seguito i risultati ottenuti effettuando una singola richiesta.

Applicazione	Richieste #	TR Medio [ms]	TR Minimo [ms]	TR Massimo [ms]	Errore %
Cloud Platform	1	55	55	55	0
Cloud Next 1	1	20	20	20	0
Cloud Next 5	1	20	20	20	0
Cloud Next 5 NGINX	1	30	30	30	0

Tabella 7.1. Test di carico singola richiesta

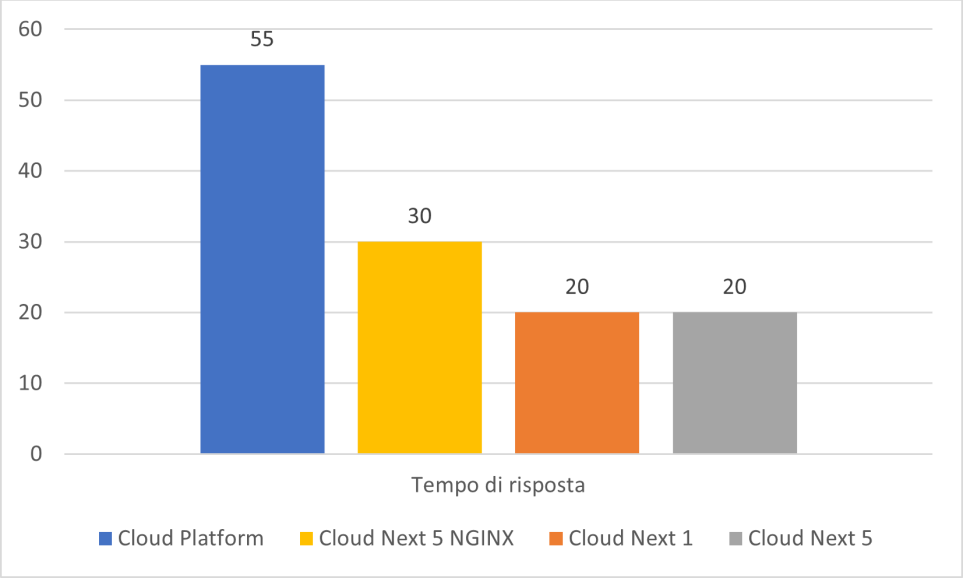

Figura 7.1. Test di carico singola richiesta

Da questo primo test possiamo osservare come la nuova applicazione risulti più veloce nella gestione di una chiamata API. Tuttavia, i benefici dati dalla scalabilità non si possono notare in quanto una singola richiesta non può che venire processata da un solo microservizio. Per questo motivo i tempi di risposta di *Cloud Next 1* e di *Cloud Next 5* coincidono. Un dato interresante è sul tempo registrato da *Cloud Next 5 NGINX*, maggiore rispetto ai due test citati sopra. Questo, probabilmente, avviene perché non si hanno benefici dall'aggiungere un livello con NGINX durante la richiesta, ma si sommano i tempi dello stesso per redirigere il traffico ad uno dei microservizi interessati.

7.3.3 1 req/sec

Di seguito i risultati ottenuti effettuando una richiesta al secondo.

Applicazione	Richieste #	TR Medio [ms]	TR Minimo [ms]	TR Massimo [ms]	Errore %
Cloud Platform	60	32	23	90	0
Cloud Next 1	60	17	14	21	0
Cloud Next 5	60	17	15	24	0
Cloud Next 5 NGINX	60	15	13	26	0

Tabella 7.2. Test di carico 1 req/sec

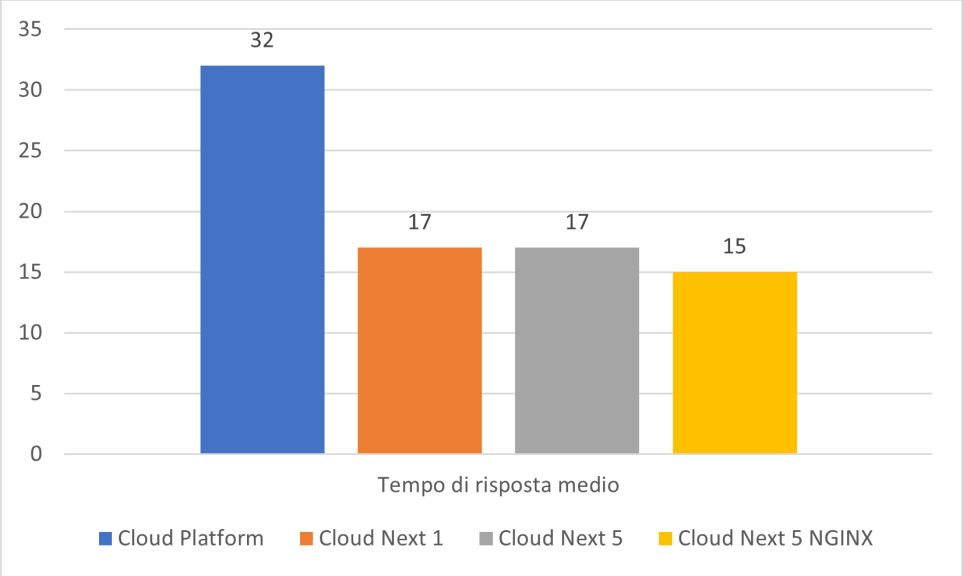

Figura 7.2. Test di carico 1 req/sec

Da questo secondo esperimento si continua a notare una superiorità in termini di prestazioni rispetto all'applicazione monolitca. Continuano a non variare di molto i dati relativi ad un singolo microservizio rispetto a cinque microservizi. Questo probabilmente è dovuto al basso rate di richieste che non consente di sfruttare più microservizi in contemporanea. Nel caso del *Cloud Platform* abbiamo un tempo di risposta massimo che si discosta di molto rispetto al tempo medio.

Per capire se ci sono dei momenti particolari in cui le richieste rallentano osserviamo la velocità di risposta nel tempo.

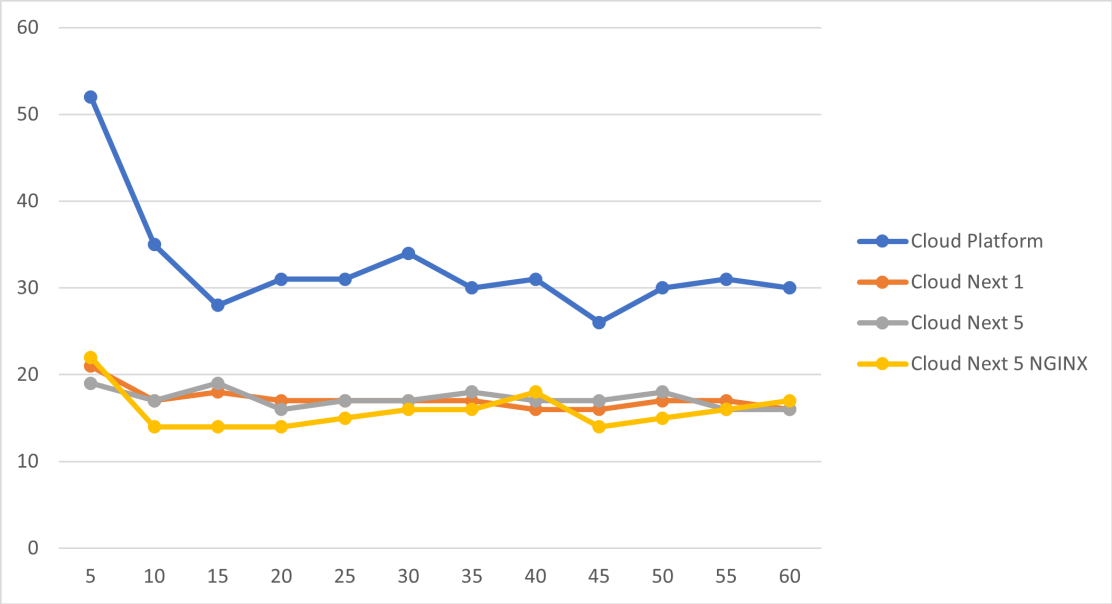

Figura 7.3. Test di carico 1 req/sec nel tempo

Si nota come il tempo maggiore di risposta si ha con le prime richieste, mentre col passare del tempo tende ad attestarsi in una fascia costante di valori.

7.3.4 10 req/sec

Di seguito i risultati ottenuti effettuando 10 richieste al secondo.

Applicazione	Richieste #	TR Medio [ms]	TR Minimo [ms]	TR Massimo [ms]	Errore %
Cloud Platform	600	24	18	139	0
Cloud Next 1	600	17	14	89	0
Cloud Next 5	600	18	15	165	0
Cloud Next 5 NGINX	600	14	11	24	0

Tabella 7.3. Test di carico 10 req/sec

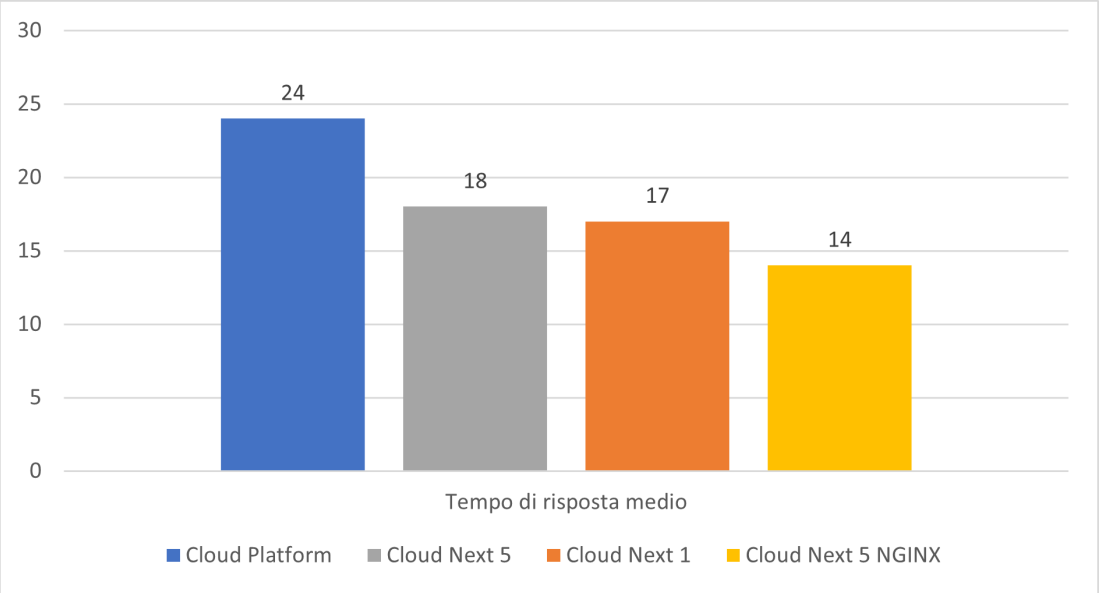

Figura 7.4. Test di carico 10 req/sec

Anche dopo questo terzo esperimento notiamo una leggera superiorità delle applicazioni a microservizi rispetto al tempo di risposta medio. Tuttavia, è necessario notare come in termini di prestazioni la differenza tra le varie applicazioni non sia così evidente. Probabilmente non siamo ancora riusciti a creare un traffico così elevato di richieste da sovraccaricare i servizi.

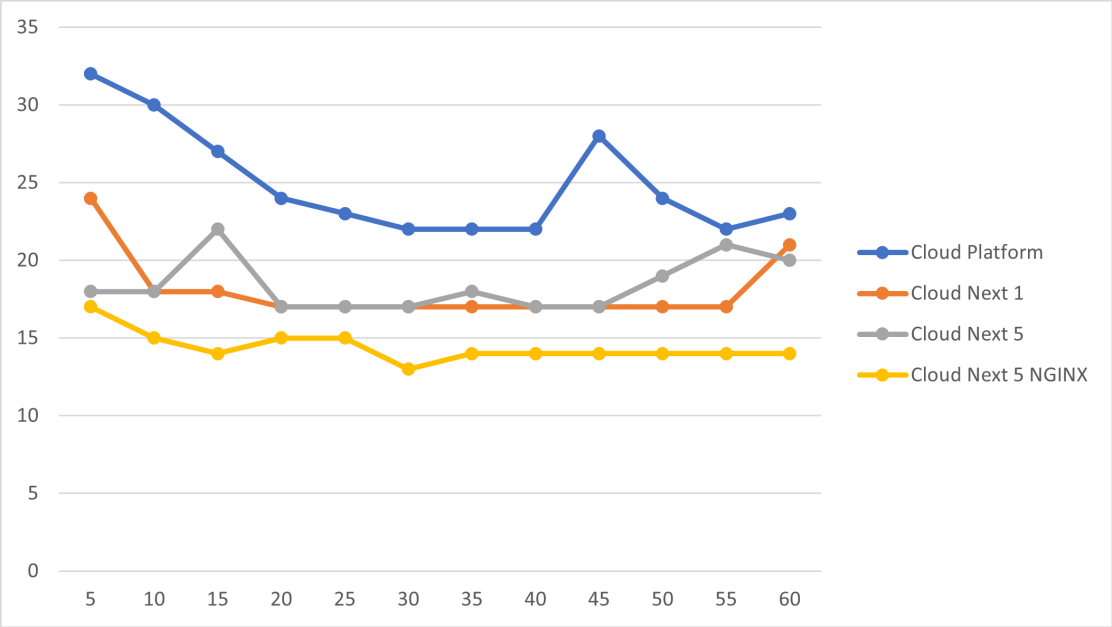

Figura 7.5. Test di carico 10 req/sec nel tempo

Osservando il grafico sulla risposta media nel tempo notiamo come l'applicazione *Cloud Next 5 NGINX* inizi a scostarsi in termini di velocità rispetto alla altre due versioni a microservizi.

7.3.5 100 req/sec

Di seguito i risultati ottenuti effettuando 100 richieste al secondo.

Applicazione	Richieste #	TR Medio [ms]	TR Minimo [ms]	TR Massimo [ms]	Errore %
Cloud Platform	6000	75346	177	144349	80.10
Cloud Next 1	6000	49151	32	62386	0
Cloud Next 5	6000	53401	29	65251	0
Cloud Next 5 NGINX	6000	39	11	795	0

Tabella 7.4. Test di carico 100 req/sec

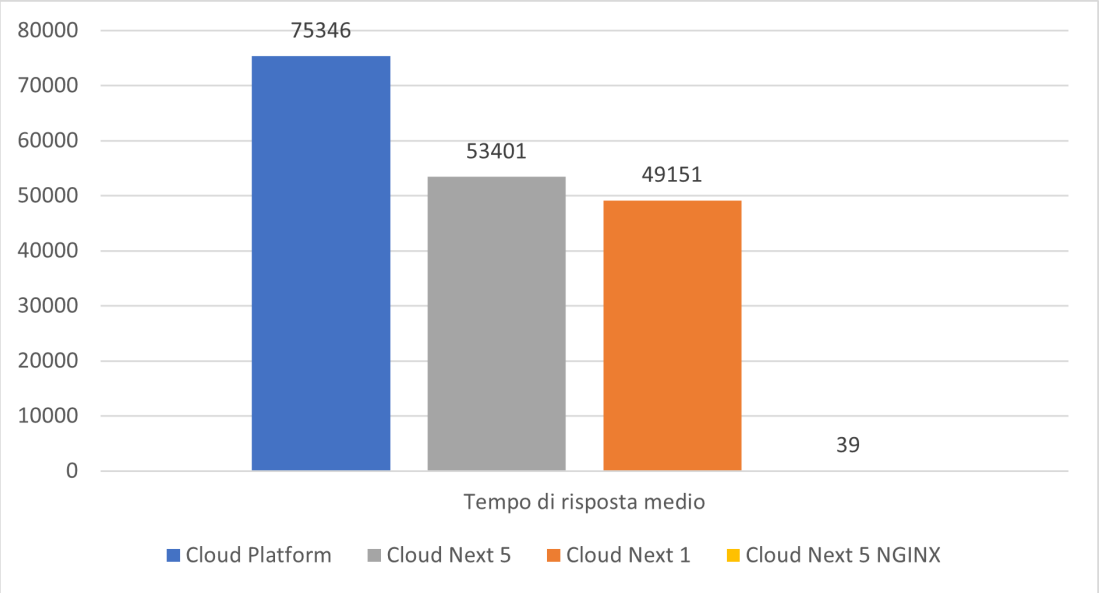

Figura 7.6. Test di carico 100 req/sec

Da quest'ultimo test emergono i risultati più interessanti. Prima di analizzare i dati è d'obbligo una precisazione: i 60 secondi della durata dell'esperimento sono intesi come tempo utile affinché si attivino tutti i thread per far partire le richieste, ma, se come in questo caso, le risposte arrivano dopo molti secondi il test può durare più a lungo.

Cloud Next 1 e *Cloud Next 5* continuano a rispondere con un andamento del tutto similare. Da questi test possiamo presupporre come il vero limite non sia dato dall'esecuzione vera e propria dell'API richiesta quanto dallo smistamento delle richieste effettuato dal servizio di *Web_Api*.

Il *Cloud Platform*, avendo al suo interno numerose funzioni in un'unico grande servizio, è l'applicazione con le prestazioni peggiori tra le quattro proposte, con un tempo medio di risposta di 75346ms. L'elemento principale di cui tenere conto è dato dalla percentuale di errore. Con l'80.10% di richieste senza esito positivo si palesa l'incapacità dell'applicazione di gestire una mole di richieste di queste dimensioni, mandando in tilt l'intero sistema di gestione delle API.

Si distingue il *Cloud Next 5 NGINX* che mantiene tempistiche di poco superiori all'esperimento precedente, confermandosi la soluzione migliore tra quelle proposte per un intenso traffico di dati.

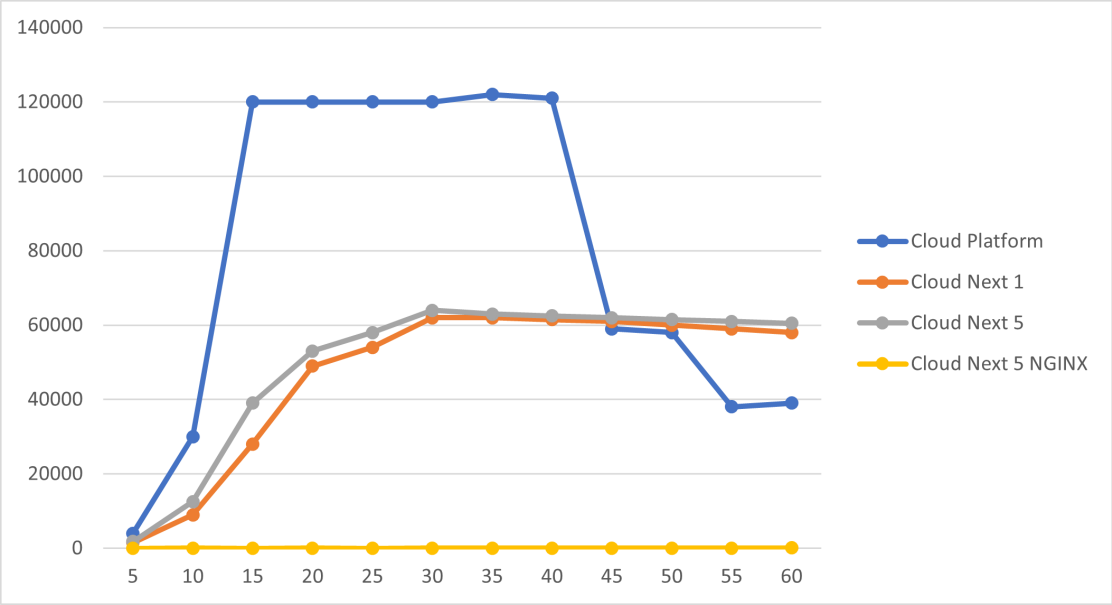

Figura 7.7. Test di carico 100 req/sec nel tempo

Da una visuale temporale dei dati si nota come il *Cloud Next 5 NGINX* non sia mai stato sotto stress, mentre le altre applicazioni a microservizi hanno incrementato gradualmente i tempi di risposta fino a mantenersi nell'ordine dei 60 secondi e proseguire con costanza fino alla fine dell'esperimento. Sebbene la velocità sia decisamente decrementata è da notare come il numero di richieste con errore sia rimasto 0.

Il *Cloud Platform* ha raggiunto i propri limiti dopo circa 15 secondi dall'inziio dell'esperimento, ma con una così alta percentuale di errore i dati sui tempi di risposta non sono attendibili ed è possibile supporre che la discesa della curva dal secondo 40 in avanti sia dovuta all'alto numero di richieste senza risposta.

7.4 Test invio dati PLC

7.4.1 Ambiente di test

Per l'analisi ho utilizzato un progetto scritto in Node.js in grado di simulare le strutture e il funzionamento di un PLC Siemens S7. Attraverso questo progetto sarò in grado di variare il valore di determinate strutture e la loro velocità di cambiamento in base ad alcune impostazioni. Per l'interfacciamento dell'applicazione

con il PLC Siemens viene utilizzata una libreria, chiamata node-snap7, in grado di leggere in polling i blocchi di dato presenti. I dati letti sono trattati come un unico array formato da interi senza segno di 8 bit l'uno che vengono successivamente parsificati in base alla configurazione decisa.

Tutti i progetti sono stati avviati durante gli esperimenti dal notebook descritto nella sezione 7.3.1

Durante questi test verranno stuidiati i tempi che intercorrono tra il cambiamento di un valore sul PLC e il salvataggio da parte dell'applicazione Cloud sul proprio database, in una tabella in cui vengono storicizzati tutti i segnali configurati. Durante i vari test i segnali varieranno i propri valori nell'arco di 60 secondi con un rate medio di variazione crescente da 1 segnale al secondo (sig/sec) fino a 50 segnali al secondo.

7.4.2 1 sig/sec

Durante questo esperimento ho fatto variare ogni secondo il valore di 1 solo segnale.

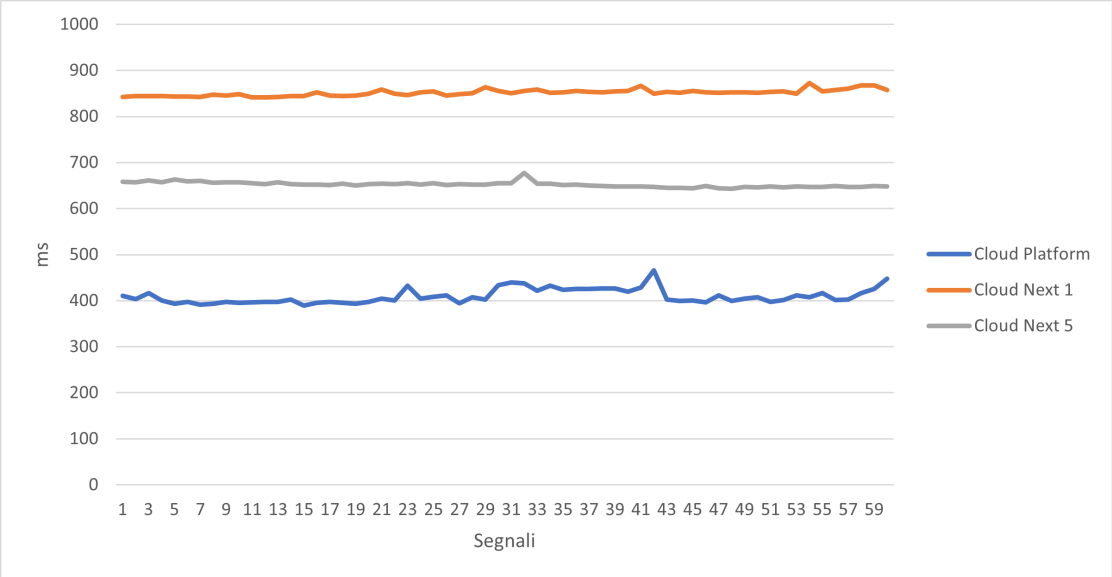

Figura 7.8. Test salvataggio 1 segnale al secondo

Osservando i valori della figura 7.8 la prima cosa che si nota è l'andamento costante dei tempi. Questo denota che le piattaforme non sono state messe in difficoltà da un carico così leggero di salvataggi da effettuare.

Si distingue in quanto a velocità l'applicazione *Cloud Platform* che, come possiamo osservare dalla figura 7.9 ha mantenuto una media di 409,45 millisecondi. Con delle medie rispettivamente di 851,12 millisecondi e di 651,98 millisecondi troviamo invece le piattaforme *Cloud Next 1* e *Cloud Next 5*.

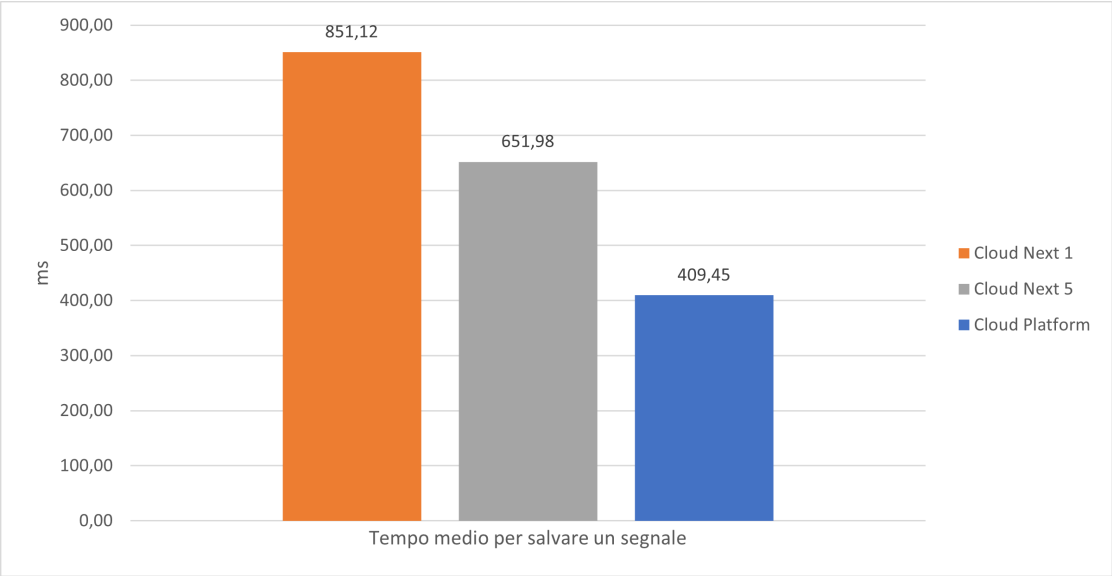

Figura 7.9. Tempo medio salvataggio 1 segnale al secondo

7.4.3 10 sig/sec

Per questo esperimento ho preferito variare 30 segnali ogni 3 secondi, mantenendo dunque una media di 10 segnali al secondo. Ho preso questa decisione per evitare di non leggere alcuni valori a causa della lettura in polling troppo stringente.

Dalla figura 7.10 iniziamo a notare alcune differenze rispetto al test precedente.

La piattaforma *Cloud Platform* che con un carico meno elevato manteneva un andamento costante, durante questo test ha dei punti di salita graduali che mostrano un discreto livello di stress.

L'applicazione *Cloud Next 1* ha continuato con un andamento costante e dei tempi di salvataggio ottimi, nonostante i 10 segnali al secondo in arrivo.

L'applicazione *Cloud Next 5* ha mantenuto dei tempi ottimi per gran parte del test, risultando per la maggior parte del tempo più veloce della corrispettiva applicazione con 1 solo servizio attivo. Tuttavia, alcuni segnali hanno impiegato più di 1 secondo ad essere salvati, facendo notevolmente aumentare il tempo medio.

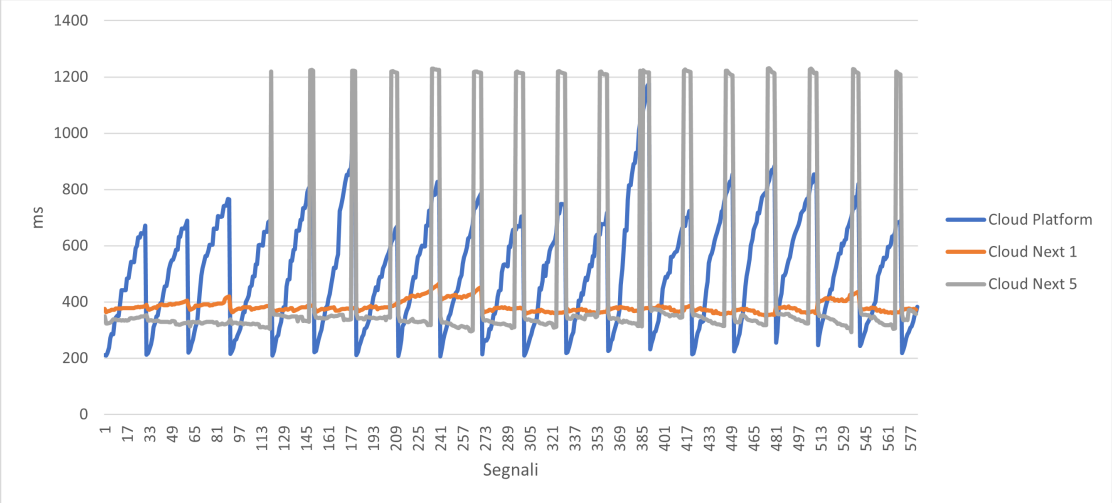

Figura 7.10. Test salvataggio 10 segnali al secondo

Osservando le medie di tutte e tre le piattaforme, 7.11, notiamo che i valori non si discostano di molto, anche se ciò che pare evidente è come l'applicazione monolitica già durante questa fase sia risultata quella più lenta delle tre.

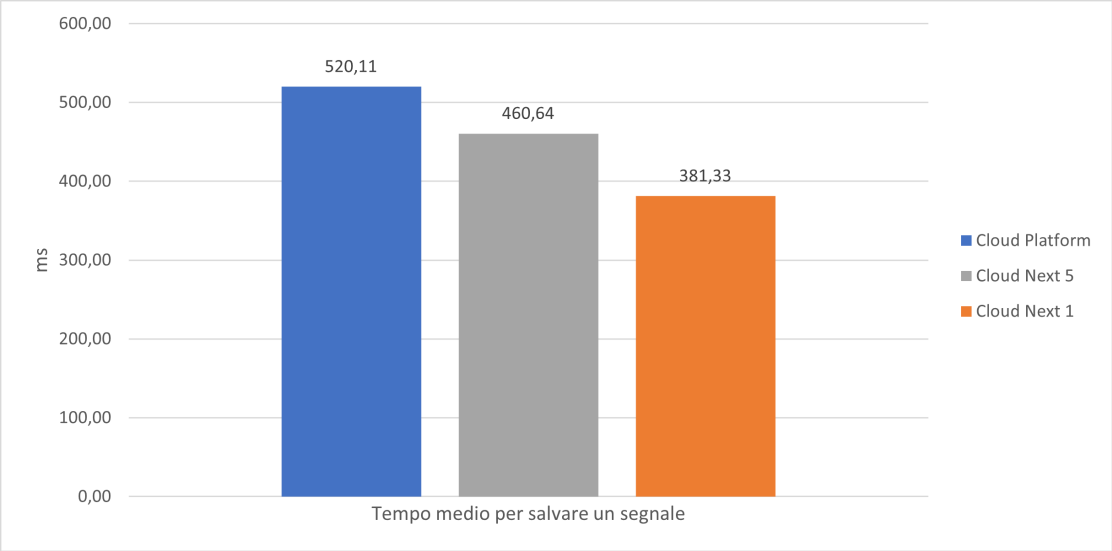

Figura 7.11. Tempo medio salvataggio 10 segnali al secondo

Dunque, al termine di questo secondo test le applicazioni a microservizi si rivelano essere leggermente più performanti rispetto a quella monolitica.

7.4.4 50 sig/sec

Per questo esperimento ho fatto variare 100 segnali ogni 2 secondi, mantenendo dunque una media di 50 segnali al secondo.

Dalla figura 7.12 notiamo dei tempi mediamente costanti per tutta la durata dell'esperimento per le applicazione a microservizi *Cloud Next 1* e *Cloud Next 5*.

L'applicazione *Cloud Platform* sotto le incessanti variazioni dei segnali da salvare si trova sicuramente sotto stress. Un dato interessante da notare è come, a differenza della figura 7.10, i tempi siano crescenti in una curva che non accenna ad arrestare la sua crescita.

La conclusione che si può trarre da questa osservazione è come il carico sia troppo per essere smaltito dal *Cloud Platform*, mentre rimanga perfettamente gestibile dalle altre due applicazioni

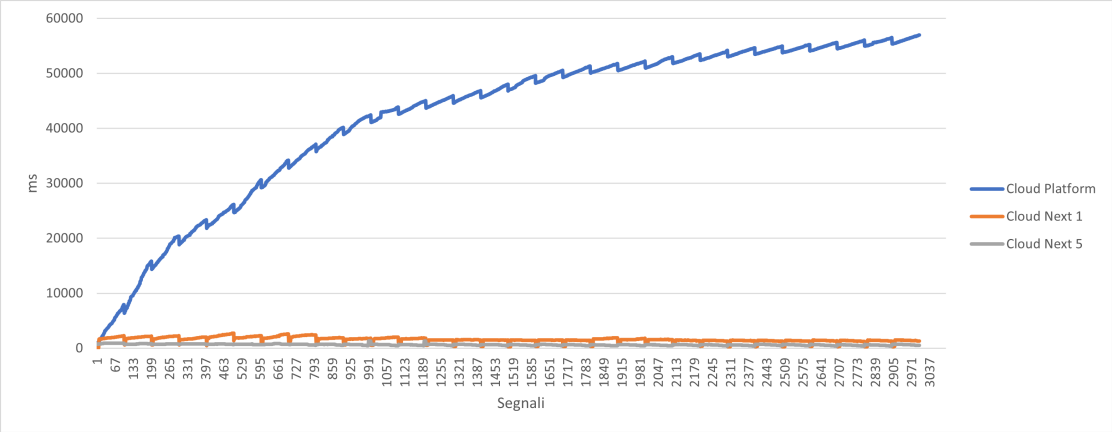

Figura 7.12. Test salvataggio 50 segnali al secondo

Osserviamo ora la figura 7.13 che mostra il tempo medio impiegato da ognuna delle tre applicazioni per il salavatggio dei dati interessati.

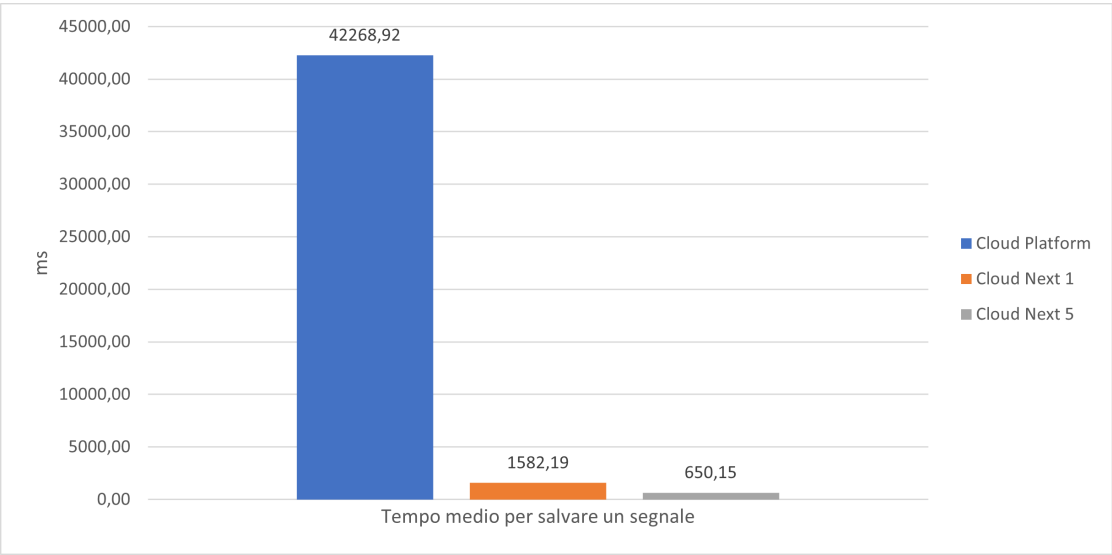

Figura 7.13. Tempo medio salvataggio 50 segnali al secondo

Dalla figura sopra notiamo un dato interessante che non emergeva dal grafico precedente. Il tempo medio registrato dal *Cloud Next 5*, 650 ms, è molto inferiore rispetto a quello del *Cloud Next 1*, 1582 ms. Questo mostra come con un alto

carico la scalabilità in 5 servizi attivi in contemporanea sia la scelta da prediligere in quanto consente di usufrire positivamente del load balancing.

Capitolo 8

Conclusioni

Al termine delle ricerche effettuate, degli sviluppi e degli esperimenti pratici si possono trarre le seguenti conclusioni.

L'applicazione monolitica scelta per i test è attualmente in uso in grosse aziende del settore Automotive e per questa ragione presenta al proprio interno numerose funzionalità che sono state aggiunte negli anni, rendendo di fatto l'applicativo un grosso e pesante blocco monolitico. Tutto questo ha sicuramente influito sui risultati dei test, che mostrano dei valori realistici e non semplici stime o supposizioni.

Dopo quattro esperimenti con carico crescente sulle API delle applicazioni ho potuto verificare i limiti dell'applicazione monolitica analizzata. Quando si prende in considerazione una piccola quantità di richieste l'applicazione monolitica e l'applicazione a microservizi hanno velocità che non differiscono di molto. Tuttavia, durante il test con un rate medio di 100 richieste al secondo, si sono viste le vere difficioltà dell'architettura monolitica. Dopo pochi secondi dall'inizio dell'esperimento la web application ha smesso di rispondere per un tempo di circa due minuti, mandando in timeout le connessioni attive con i database e rendendo inutilizzabile sia l'interfaccia web, messa a disposizione degli utenti, che il resto delle funzionalità presenti nell'applicativo. Oltre a questa interruzione dei servizi, le richieste hanno ricevuto come risposta un messaggio di errore per circa l'80% dei casi rendendo palesi le difficoltà riscontrate dal sistema.

Parlando, invece, dell'applicazione a microservizi la percentuale di errore è rimasta fissa sullo 0% per tutta la durata dei test denotando un alto livello di affidabilità. I tempi di risposta sono incrementati notevolmente durante l'ultimo esperimento, ma proprio per la tipologia di questa architettura gli altri servizi dell'applicazione hanno continuato a svolgere le proprie funzionalità senza essere danneggiati dall'alto carico di richieste ricevute. Uno dei limiti osservati è nella velocità di smistamento delle richieste arrivate al servizio di Web_Api. Per superare questo

ostacolo ho deciso di utilizzare NGINX, come load balancer tra i vari nodi, consentendomi così di istanziare più servizi di Web_Api e gestire in maniera fluida lo smistamento delle richieste arrivate. Con quest'ultima configurazione le prestazioni sono rimaste invidiabili persino con un rate medio pari a 100 richieste al secondo.

Dopo tre esperimenti di invio dati da PLC, con numero sempre crescente di informazioni da salvare, ho ottenuto dei risultati interessanti. L'applicazione monolitca con un carico di 1 dato al secondo da salvare rimane performante e con tempi addirittura migliori rispetto alle concorrenti applicazioni a microservizi. Tuttavia, al crescere dei segnali l'applicazione raggiunge uno stato di stress tale da non riuscire più a smaltire i dati in arrivo in tempo reale e continua ad accumulare ritardo.

L'applicazione a microservizi ha mantenuto un andamento costante, nonostante il carico crescente. Ha registrato un incremento del tempo medio, intercorso dalla lettura del segnale fino al suo salvataggio, riuscendo comunque a smaltire tutti i segnali inviati in tempi ragionevoli. Con l'utlimo esperimento si è anche notato un netto miglioramento nell'utilizzo di 5 servizi attivi contemporanemante e nella possibilità di bilanciare i dati da salvare tra questi microservizi.

Dunque, facendo delle considerazioni finali, ritengo che per applicazioni di piccole dimensioni e con una complessità non elevata i benefici che si possono trarre da un'architettura a microservizi non siano rilevanti. Mentre, al crescere della complessità e del bacino d'utenza, può essere conveniente investire in questa tipologia di architettura, migliorando in questo modo la leggibilità del codice, la manutenzione, la velocità nella creazione delle nuove release e le prestazioni.

Bibliografia

[1] What is ci/cd? URL https://www.redhat.com/en/topics/devops/what-is-ci-cd. Ultimo accesso: 19/06/2022.

[2] Cos'è devops?, . URL https://aws.amazon.com/it/devops/what-is-devops/. Ultimo accesso: 19/06/2022.

[3] I vantaggi dell'approccio devops, . URL https://www.redhat.com/it/topics/devops#panoramica. Ultimo accesso: 19/06/2022.

[4] Che cos'è dev ops?, . URL https://edalab.it/che-cose-dev-ops/. Ultimo accesso: 19/06/2022.

[5] Docker overview. URL https://docs.docker.com/get-started/overview/. Ultimo accesso: 24/06/2022.

[6] Express. URL https://www.npmjs.com/package/express. Ultimo accesso: 24/06/2022.

[7] i18n. URL https://www.npmjs.com/package/i18n. Ultimo accesso: 24/06/2022.

[8] Apache jmeter. URL https://jmeter.apache.org/. Ultimo accesso: 02/07/2022.

[9] Knockout introduction. URL https://knockoutjs.com/documentation/introduction.html. Ultimo accesso: 24/06/2022.

[10] What are the benefits of microservices architecture? URL https://www.appdynamics.com/topics/benefits-of-microservices. Ultimo accesso: 19/06/2022.

[11] What is moleculer? URL https://moleculer.services/docs/0.14/. Ultimo accesso: 25/06/2022.

[12] What is nats. URL https://docs.nats.io/nats-concepts/what-is-nats. Ultimo accesso: 25/06/2022.

[13] Introduction to node.js. URL `https://nodejs.dev/learn/introduction-to-nodejs`. Ultimo accesso: 24/06/2022.

[14] Redis. URL `https://redis.io/`. Ultimo accesso: 24/06/2022.

[15] Sequelize. URL `https://sequelize.org/`. Ultimo accesso: 24/06/2022.

[16] What socket.io is. URL `https://socket.io/docs/v4/`. Ultimo accesso: 24/06/2022.

[17] Nicolas Bohorquez. Getting started with javascript and influxdb. URL `https://thenewstack.io/getting-started-with-javascript-and-influxdb/`. Ultimo accesso: 25/06/2022.

[18] geekandjob. 20 framework node.js da usare nel 2022, . URL `https://blog.geekandjob.com/framework-node-js/`. Ultimo accesso: 25/06/2022.

[19] geekandjob. Cos'è nuxt.js, . URL `https://www.geekandjob.com/wiki/nuxt-js`. Ultimo accesso: 25/06/2022.

[20] geekandjob. Cos'è vue, . URL `https://www.geekandjob.com/wiki/vue`. Ultimo accesso: 25/06/2022.

[21] kinsta. Cosa è nginx? una rapida occhiata a cosa è e come funziona. URL `https://kinsta.com/it/knowledgebase/cosa-e-nginx/`. Ultimo accesso: 02/07/2022.

[22] Luciana Maci. Che cos'è l'industria 4.0 e perché è importante saperla affrontare. URL `https://www.economyup.it/innovazione/cos-e-l-industria-40-e-perche-e-importante-saperla-affrontare/`. Ultimo accesso: 26/06/2022.

[23] Giuseppe Maggi. Node.js. URL `https://devacademy.it/node-js/`. Ultimo accesso: 24/06/2022.

[24] Technical matters. Influxdb – explanation, advantages, and first steps. URL `https://www.ionos.com/digitalguide/hosting/technical-matters/what-is-influxdb/`. Ultimo accesso: 25/06/2022.

[25] redazione di ZeroUno. Osservatorio iot 2022, incrementi del mercato di oltre il 20%. URL `https://www.zerounoweb.it/trends/dinamiche-di-mercato/osservatorio-iot-2022-incrementi-del-mercato-di-oltre-il-20/`. Ultimo accesso: 26/06/2022.

[26] MW Team. What are microservices? how microservices architecture works. URL `https://middleware.io/blog/microservices-architecture/`. Ultimo accesso: 19/06/2022.

[27] Francesco La Trofa. Cos'è e come funziona l'architettura monolitica. URL `https://universeit.blog/architettura-monolitica/`. Ultimo accesso: 19/06/2022.

[28] Siraj ul Haq. Monolithic architecture. URL `https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63`. Ultimo accesso: 19/06/2022.