POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Valutazione dei vantaggi di applicazioni a microservizi rispetto ad applicazioni monolitiche su un caso d'uso reale

Relatore:

Prof. Riccardo Sisto

Candidato:

Paolo Bello

ANNO ACCADEMICO 2018/2019

Sommario

1 INTRODUZIONE

In previsione del triennio 2018-2020 TIM ha presentato un piano strategico per consolidare la sua posizione nel mercato: DigiTIM.

DigiTIM punta sull'utilizzo di quelle che probabilmente saranno le tecnologie più rilevamenti nei prossimi anni: cloud, big data, Internet of Things, 5G. Mira a snellire i processi aziendali, a semplificare e velocizzare lo sviluppo, a digitalizzare l'interazione con i clienti sfruttando l'intelligenza artificiale ed integrando i servizi in un'unica piattaforma.

Un elemento centrale di DigiTIM è il settore IT, che mira a virtualizzare le infrastrutture sfruttando la potenza del cloud ed introducendo nuove metodologie di sviluppo.

Uno degli obiettivi prefissati del settore IT è quello di rivoluzionare i servizi offerti ai clienti, sostituendo le attuali applicazioni monolitiche con delle applicazioni a microservizi installate su piattaforme cloud. È proprio questo il punto focale del mio studio, nel quale ho strutturato delle possibili soluzioni per gestire in modo efficiente e scalabile una tematica molto rilevante per un'azienda di telecomunicazioni leader del mercato: l'invio massivo di comunicazioni ad un grande numero di utenti, tramite i classici canali di comunicazione (sms, notifiche push tramite app, e-mail), attualmente demandato ad un sistema monolitico.

Per analizzare queste soluzioni occorre conoscere a fondo il mondo dello sviluppo a microservizi, la metodologia, le tecniche e le tecnologie, nonché i principali pattern architetturali.

Inizierò quindi introducendo la metodologia DevOps, posta alla base dello sviluppo di microservizi. In seguito, nel corso della trattazione esporrò la mia proposta, spiegando dettagliatamente le scelte effettuate e cercando di analizzarne i vantaggi. In conclusione, illustrerò un'analisi prestazionale che permetta di avere una panoramica completa della mia soluzione.

2 METODOLOGIA DEVOPS

Il termine DevOps nasce dalla contrazione di due termini, *Development* ed *Operations*. Concerne, infatti, una stretta correlazione tra la fase "Dev", quella di pianificazione e sviluppo, ed "Ops", quella dedicata al rilascio e, più in generale, al ciclo di vita del software. Tale correlazione mira a rendere lo sviluppo ed i rilasci di software molto più rapidi e produttivi.

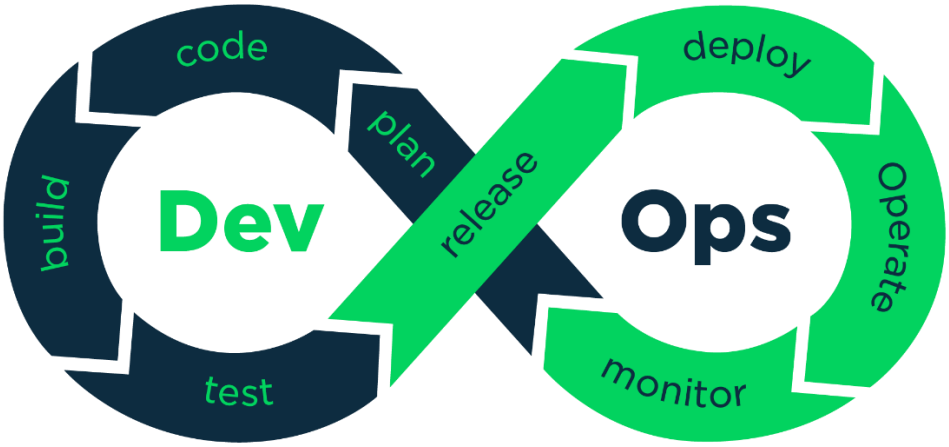

Figura 2.1 – Principali fasi di Development e IT Operations [1]

2.1 LA NASCITA DEL DEVOPS

La nascita della metodologia DevOps risale intorno al 2009, quando John Allspaw e Paul Hammond, due dipendenti di Flickr.com, in seguito alla necessità della loro azienda di effettuare numerosi rilasci di software al giorno, presentarono in occasione del O'Reilly Velocity Conference "10+ Deploys Per Day: Dev and Ops Cooperation at Flickr", illustrando i continui problemi dell'interazione tra lo sviluppo di applicazioni e le attività operative ed evidenziando come queste parti potessero essere perfettamente integrate.

Nello stesso anno, l'informatico Patrick Debois, dopo aver assistito alla presentazione in Web Streaming, decise di indire una conferenza

sull'argomento, organizzando i "*DevOps Days"* in Belgio. Fu proprio lui a coniare il termine DevOps, lanciando l'hashtag per pubblicizzare l'evento.

Nel 2010 vennero organizzati per la prima volta i *DevOps Days* negli Stati Uniti, precisamente a Mountain View, in California, in occasione dell'annuale Velocity Conference. Ebbero molto successo, per cui vennero ripetuti molte altre volte fino ad oggi: basti pensare che solo nel 2018 sono stati organizzate più di trenta conferenze DevOps Days, molte delle quali si svolgono negli Stati Uniti.

L'approccio DevOps ha iniziato ad essere sempre più applicato nelle aziende, dove i team dedicati allo sviluppo di codice e quelli dedicati alla produzione di software non agiscono più separatamente, ma fanno parte di un'unica unità che segue l'intero ciclo di vita dell'applicazione, principalmente grazie all'automatizzazione di gran parte dei processi.

2.2 I VANTAGGI

- *Velocità:* i processi sono molto più snelli ed agili, e permettono di tenere il passo delle sempre più esigenti richieste di mercato.
- *Automatizzazione dei processi:* questa diminuisce notevolmente la complessità di molte operazioni critiche durante un rilascio, come ad esempio build, test e deploy.
- *Efficienza:* è una diretta conseguenza dei primi due punti. Grazie alla rapidità di sviluppo e all'automatizzazione, lo sviluppo software risulta di gran lunga più efficiente. Il rilascio di nuove release avviene in maniera immediata, rendendo possibile la frequente pubblicazione di aggiornamenti.
- *Scalabilità:* l'approccio in questione offre dei metodi per gestire sistemi e processi su qualsiasi scala. Sfruttando la stessa infrastruttura sarà possibile adattare le risorse allocate in tempo reale *"on demand"*, in base al numero di richieste ricevute.
- *Affidabilità:* grazie all'integrazione continua, sulla quale approfondirò più avanti, è possibile verificare costantemente la qualità degli aggiornamenti e dei sistemi. È possibile monitorare continuamente le prestazioni tramite dei sistemi di log.
- *Collaborazione:* interazione continua tra sviluppatori e sistemisti, i quali si confronteranno su tutte le fasi del ciclo di vita del software.

- *Sicurezza:* la velocità di sviluppo non va assolutamente a discapito della sicurezza. La sicurezza deve diventare parte integrante di tutte le fasi del ciclo di vita, dal principio alla fine dello sviluppo. Alcune volte, proprio per sottolineare l'importanza della sicurezza, anziché *DevOps* qualcuno usa il termine *"DevSecOps"* (Development, Security, Operations).

2.3 PRINCIPI DELLA METODOLOGIA DEVOPS

Lo sviluppo DevOps si fonda su alcuni principi riguardanti sia il design che il rilascio di applicazioni.

Innanzitutto, riferiamoci all'architettura del software: come tratterò ampiamente più avanti, le classiche architetture monolitiche non si prestano assolutamente allo sviluppo DevOps. Seguendo il famoso paradigma informatico *"divide et impera"* per usufruire dei vantaggi di questa metodologia occorre partizionare le architetture, cercando di crearne una che offra le stesse funzionalità dell'applicazione monolitica, mettendo però a disposizione molteplici servizi indipendenti. Solitamente si tratta di porzioni di codice di piccole dimensioni, atte a svolgere un singolo compito: in quest'ultimo caso non si parla di servizi, ma di microservizi. "Ogni microservizio esegue il proprio processo e comunica con gli altri servizi tramite un'interfaccia predefinita utilizzando un mezzo con impatto ridotto sulle prestazioni, solitamente una API (Application Programming Interface) basata su http" [2].

Qualsiasi applicazione deve avere un sistema che ospita il proprio codice, un'infrastruttura sulla quale l'applicazione viene installata e resa disponibile. L'infrastruttura che meglio sposa i principi DevOps è la cosiddetta *"infrastruttura come codice"*. "Si tratta di una prassi secondo cui provisioning e gestione dell'infrastruttura avvengono tramite metodologie di sviluppo di software e codice. Questo tipo di infrastruttura favorisce il controllo di versione e l'integrazione continua" [3].

Il concetto di *"Integrazione Continua"* (dal termine inglese Continuous Integration) si basa sul rilascio continuo di codice, che va appunto ad integrare tempestivamente quanto già rilasciato.

Infine, un altro aspetto importante è quello del mantenimento del codice. Lo sviluppo DevOps mette a disposizione dei sistemi di registrazione e

consultazione di log, indispensabili per scovare reattivamente errori o imperfezioni nel codice e rilasciare le opportune modifiche.

Nei prossimi paragrafi, dopo aver brevemente approfondito il concetto di *Integrazione Continua*, tratterò più ampiamente i principi DevOps sul design di architettura e infrastruttura appena esposti, illustrando a fondo il modello di *Architettura a Microservizi* e di *Infrastruttura come Codice*, e, durante la trattazione, illustrerò le principali scelte implementative per lo studio alla base di questa ricerca, cercando di utilizzare le tecnologie più calzanti al mio scopo.

3 INTEGRAZIONE CONTINUA

Per *Integrazione Continua* si intende un metodo di sviluppo che mira a rilasciare continuamente nuove versioni del Software, atte a correggere tempestivamente eventuali errori nel codice ed a favorire la rapidità di pubblicazione di nuovi aggiornamenti.

È una pratica ad oggi molto comune all'interno di Team con diversi sviluppatori che lavorano sullo stesso progetto, uno scenario ormai molto comune. Sarebbe ovviamente molto dispendioso per uno sviluppatore apportare delle modifiche ad un progetto tenendo in considerazione eventuali altre modifiche effettuate da altri. È quindi indispensabile contestualmente alla fase di modifica preoccuparsi anche del rilascio, per poter mettere a disposizione di un altro membro del Team sempre una versione aggiornata.

L'integrazione continua avviene grazie all'automatizzazione delle fasi di build, test e deploy, che abbatte drasticamente i tempi necessari per effettuare un rilascio.

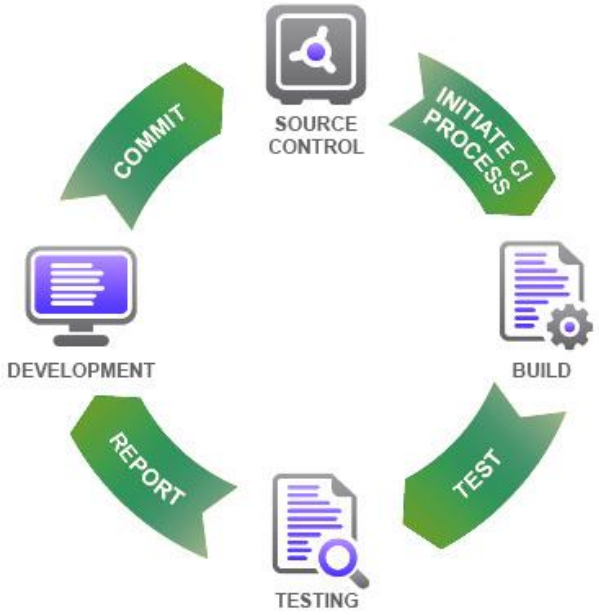

Figura 3.1 - Ciclo dell'integrazione continua [4]

Per applicare il metodo di Integrazione continua è necessario disporre di un repository centralizzato dove il software viene di volta in volta rilasciato, e di un sistema di versioning che permetta di riconoscere le diverse versioni del software dopo i vari rilasci.

Ad ogni rilascio sul repository il software può essere sottoposto a dei test automatizzati.

Il corretto utilizzo di un modello ad *Integrazione Continua* porta molti dei vantaggi comuni alla metodologia DevOps, ossia:

- Velocità nel rilascio di aggiornamenti (bux fix, nuove funzionalità, ecc.).
- Identificazione di errori o debolezze del codice a monte del rilascio (grazie ai test automatici a cui si sottopone la nuova versione sviluppata).
- Maggiore produttività.

4 ARCHITETTURA MONOLITICA ED ARCHITETTURA A MICROSERVIZI

4.1 APPLICAZIONI MONOLITICHE

Un'applicazione si definisce monolitica quando viene sviluppata e distribuita come una singola entità ed i suoi componenti fanno parte di un singolo programma su una singola piattaforma. Ciò significa che, dietro la UI (User-Interface), è presente un unico grande applicativo responsabile di tutto, dalla fase di autenticazione degli utenti alla logica applicativa, e un unico grande database.

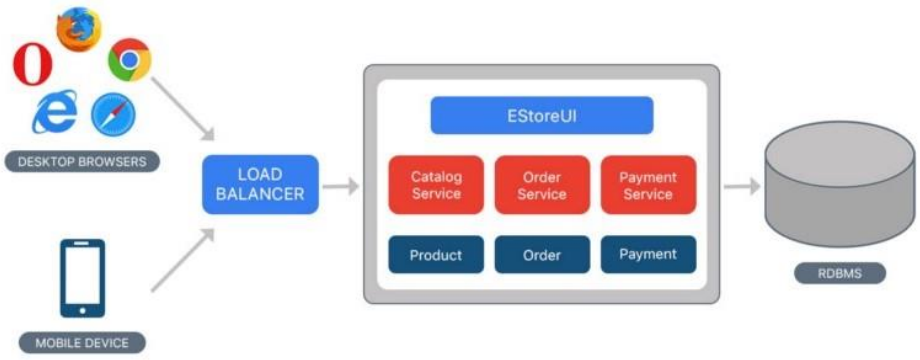

Figura 4.1 - Esempio di architettura monolitica per un'applicazione di e-commerce [5]

Questo modello di sviluppo è sicuramente più sbrigativo, e può funzionare bene per applicazioni di dimensioni contenute o per applicazioni poco soggette a cambiamenti. Ma nel caso in cui dovessimo sviluppare applicazioni complesse e che mutano nel tempo, come gran parte delle attuali richieste in ambito web e mobile, allora questa architettura raggiungerebbe dimensioni mastodontiche, rendendo sempre più difficile la manutenzione del codice dopo i primi rilasci.

4.2 PROBLEMI DELL'ARCHITETTURA MONOLITICA

Come accennato in precedenza, l'approccio tramite architettura monolitica può portare in alcuni casi dei vantaggi, che possono essere riassunti in un unico termine: semplicità. Un'applicazione monolitica può sembrare semplice da sviluppare, rilasciare e scalare (semplicemente istanziando più volte lo stesso applicativo).

Ma al crescere dell'applicazione, ecco comparire una serie di problematiche:

- il concetto di integrazione continua è molto difficile da applicare, un applicativo molto grande sfavorisce i rilasci frequenti, rendendoli lenti e macchinosi.
- è difficile conoscere a fondo un enorme blocco di codice. Ci si potrebbe facilmente trovare nella condizione in cui nessuno sviluppatore comprenda l'interezza dello sviluppo dell'applicazione.
- un'applicazione molto grande può scoraggiare nuovi sviluppatori che devono approcciarsi allo sviluppo di correzioni o aggiornamenti, in quanto dovranno comprendere pienamente tutto (o quasi) il codice anche per apportare piccole modifiche.
- il riutilizzo dell'applicazione è molto limitato.
- il container dell'applicazione o l'ambiente di sviluppo potrebbero essere rallentati dalla quantità di codice.
- è molto difficile rendere l'applicazione scalabile, andando quasi sempre incontro ad uno spreco di risorse.

Vedremo come risolvere queste problematiche scalando una applicazione monolitica, suddividendola in un'architettura a microservizi.

4.3 THE SCALE CUBE

Tra le varie strategie per scalare una tra le più note si basa su un modello tridimensionale, lo *"scale cube"*. Questo modello spiega come far scalare un'applicazione lungo i suoi tre assi, associando dei metodi di *"scaling"* ai movimenti lungo gli assi di un cubo.

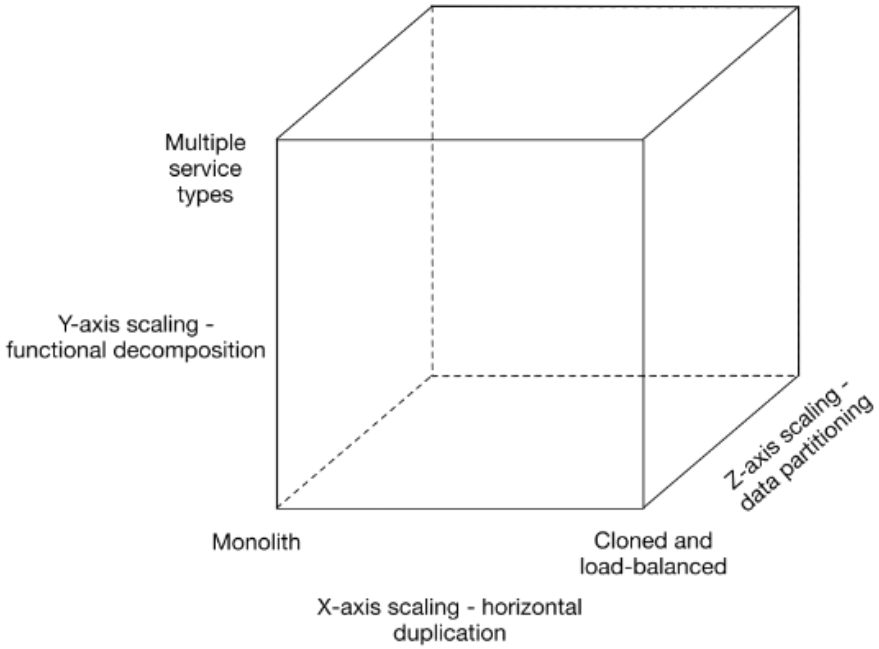

Figura 4.2 - Rappresentazione dello "scale cube" [6]

L'approccio più semplice consiste nello scalare un'applicazione lungo l'asse X. In questo caso si effettua il deploy di più istanze della stessa applicazione, le quali vengono gestite da un *load balancer*. È il metodo più comune per scalare un'applicazione monolitica, ma è anche il più inefficiente, soprattutto in caso di applicazioni di grandi dimensioni.

Un approccio simile è quello di scalare lungo l'asse Z. Anche in questo caso vengono effettuati deploy multipli, ma in questo caso ognuno di essi è istanziato da un server differente, che opera solo su un suo sottoinsieme di dati. Sebbene questo approccio apporti delle migliorie, sono ancora presenti i problemi legati alla complessità per grandi applicazioni.

Il terzo approccio, lungo l'asse Y, consiste nel dividere l'applicazione in più servizi, diversi tra loro, ognuno dei quali responsabile di una funzione. In questo caso non esistono problemi di complessità, dovendo gestire piccole porzioni di codice indipendenti.

La tecnica di scaling più efficace risulta una combinazione tra l'asse Y e l'asse Z, dividendo l'applicazione in più microservizi, ciascuno responsabile dei propri dati ed istanziato all'occorrenza.

4.4 ARCHITETTURA A MICROSERVIZI

Un'architettura a microservizi permette di dividere un'applicazione in differenti servizi indipendenti, atti a svolgere un'unica funzionalità di business e aventi delle interfacce per comunicare tra loro. [7]

Figura 4.3 - Esempio di architettura a microservizi per un'applicazione di e-commerce [8]

Invece di utilizzare un unico grande database, in questa architettura ogni servizio ha il proprio, strutturato secondo le proprie esigenze. "Non è necessario che i servizi condividano lo stesso stack di tecnologie, le stesse librerie o gli stessi framework" [9]. Ognuno di essi, infatti, non offre visibilità agli altri, ma come detto in precedenza espone delle API (Application Programming Interface) per comunicare con l'esterno.

4.5 VANTAGGI DELL'ARCHITETTURA A MICROSERVIZI

Il passaggio all'architettura a microservizi introduce diversi vantaggi:

- sviluppo più agile e rapido che favorisce l'integrazione continua. Un singolo team di sviluppo può sviluppare, modificare, testare e rilasciare un microservizio, scegliendo indipendentemente il framework o il linguaggio che reputa più opportuno. Ne consegue un ritmo di rilascio più veloce.
- è possibile effettuare correzioni di bug o aggiornamenti contestualizzati alla singola funzione, senza dover ridistribuire l'intera applicazione.

- riusabilità: trattandosi di servizi mirati ad una singola funzione di business, possono essere riciclati e riutilizzati sia all'interno dell'applicazione stessa che in altri contesti.
- la scalabilità è notevolmente migliorata, grazie al fatto di poter istanziare in seguito a reali necessità di un servizio, allocando risorse "on demand" in maniera elastica.
- individuazione errori ed eliminazione del single-point-of-failure: grazie a questa architettura distribuita, viene eliminato l'elemento centrale che poteva essere un single-point-of-failure. Separando e isolando i servizi l'individuazione di errori diviene più semplice ed efficace.

4.6 INFRASTRUTTURE PER MICROSERVIZI

Una volta sviluppato, dove risiede un microservizio? Un microservizio, in fase di installazione ed esecuzione, richiede all'ambiente in cui risiede molte tecnologie e configurazioni per trarre vantaggio dalle sue caratteristiche. Ma configurare in modo corretto un'infrastruttura in modo classico può essere molto dispendioso e richiedere la partecipazione di una figura specializzata, come un sistemista. Inoltre questo, essendo un approccio "manuale", si presta particolarmente all'introduzione di errori o mal-configurazioni.

Assolutamente in linea con la metodologia DevOps è nata la cosiddetta *Infrastruttura come Codice*, per ovviare a questi problemi e cercare di semplificare la configurazione del sistema ospitante l'applicazione.

5 INFRASTRUTTURA COME CODICE

Il concetto di *IaC* (Infrastructure as Code) nasce come detto in contrapposizione alla pratica di configurazione manuale delle infrastrutture.

Come suggerisce il nome, si basa sull'idea di sviluppare ed utilizzare del software che gestisca la configurazione e la messa in funzione dell'infrastruttura alla base di applicazioni. Questo approccio rende l'infrastruttura stessa più elastica e flessibile, ed aggiunge due qualità: ripetibilità e scalabilità.

L'uso di *Infrastructure as Code* permette di creare un'infrastruttura in maniera più semplice e di ridurre il numero di errori in fase di configurazione, soprattutto nei casi in cui si abbia bisogno di riconfigurarla frequentemente. Supporta pienamente l'idea di infrastruttura cloud, non diversa dalla precedente se non per il fatto di essere virtualizzata ed offerta come servizio e che, proprio per questo motivo, prende il nome di IaaS (Infrastructure as a Service).

"IaaS costituisce la base del cloud computing e consente agli utenti di realizzare piattaforme IT virtualizzate facilmente scalabili" [10]. Oltre a IaaS, esistono altri tipi di infrastrutture fruite come servizi cloud, come PaaS (Platform as a Service) e SaaS (Software as a Service).

5.1 IL MERCATO DEI SERVIZI CLOUD

Nel mondo dello sviluppo software le soluzioni basate sul cloud sono sempre più frequenti.

IDC (International Data Corporation), azienda mondiale specializzata in ricerche di mercato, ha realizzato nel 2018 una statistica che riguarda gli investimenti delle società IT a livello mondiale [11]. Il punto focale della ricerca è il rapporto tra le spese per i tradizionali Data Center e per le infrastrutture cloud.

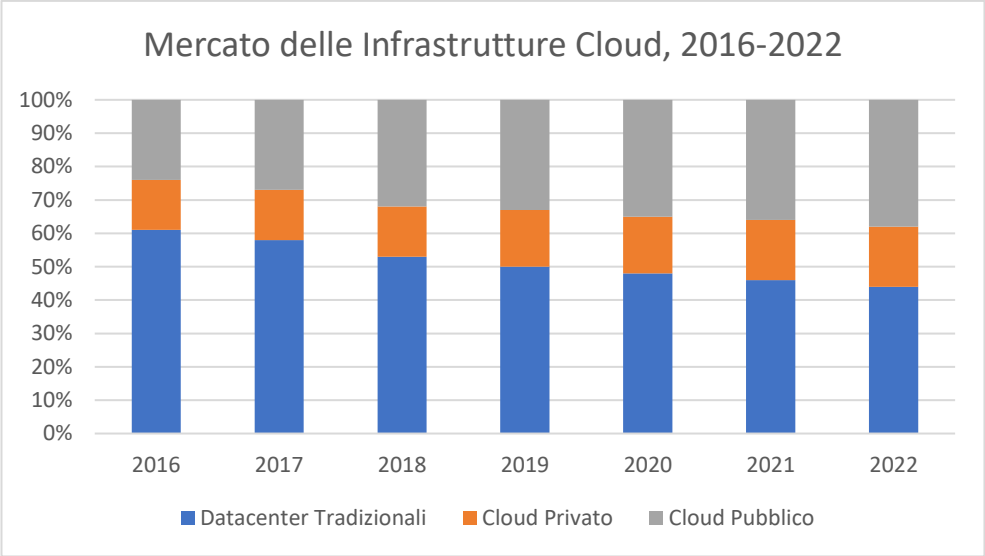

Grafico 5.1 - Rappresentazione degli investimenti delle aziende IT per soluzioni cloud e tradizionali

Dal Grafico 5.1 si evince come nel corso degli anni il rapporto degli investimenti tra soluzioni cloud e tradizionali sia in continuo aumento. IDC ha previsto che nel 2022 la spese dedicate ai servizi cloud saranno circa il 56% del totale delle spese per le infrastrutture IT.

Un'altra ricerca di Aprile 2018 effettuata da Gartner, società statunitense leader mondiale nella consulenza, ricerca e analisi in ambito IT, evidenzia l'incremento del mercato dei servizi cloud, in termini di miliardi di dollari [12].

Lo studio, presentato nella Tabella 1, è stato effettuato considerando le più importanti tipologie di servizi di infrastrutture cloud, ossia:

- SaS (Software as a Service)
- BpaaS (Business Process as a Service)
- IaaS (Infrastructure as a Service)
- PaaS (Platform as a Service)

	2017	**2018**	**2019**	**2020**	**2021**
Cloud Business Process Services (BPaaS)	42.6	46.4	50.1	54.1	58.4
Cloud Application Infrastructure Services (PaaS)	11.9	15.0	18.6	22.7	27.3

Cloud Application Services (SaaS)	60.2	73.6	87.2	101.9	117.1
Cloud Management and Security Services	8.7	10.5	12.3	14.1	16.1
Cloud System Infrastructure Services (IaaS)	30.0	40.8	52.9	67.4	83.5
Total Market	**153.4**	**186.3**	**221.1**	**260.2**	**302.4**

Tabella 5.2 - Previsione dei ricavi nel mercato mondiale dai servizi cloud (in miliardi di dollari)

Il dato più rilevante è certamente l'incredibile aumento dei volumi di mercato in un intervallo di soli 5 anni, e, come deducibile dal grafico in Figura 7, gli investimenti per PaaS e IaaS supereranno il doppio del valore riferito al 2017.

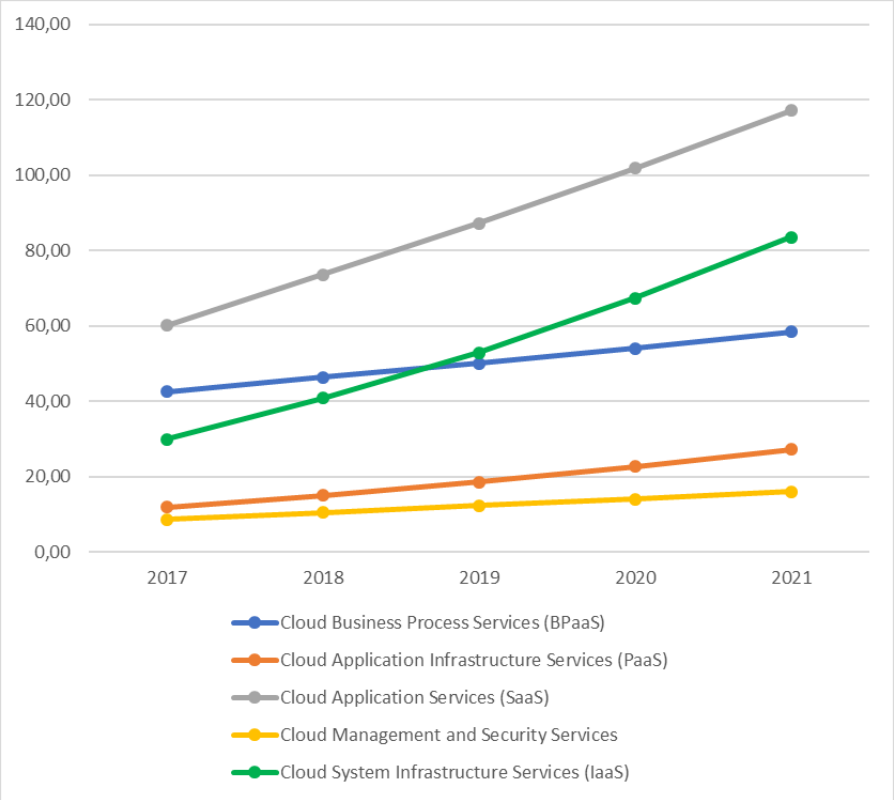

Grafico 5.3 – Rappresentazione volumi di mercato indicati in Tabella 5.2

Nonostante i volumi di mercato importanti, non tratterò i servizi cloud di Business Process (BPaaS), ma mi concentrerò sui primi tre strati, IaaS, PaaS e SaaS.

5.2 IaaS, PaaS e SaaS

IaaS, PaaS e SaaS: tutti e tre i modelli portano dei vantaggi derivati dal mondo dei servizi cloud. Concettualmente possiamo rappresentarli secondo uno schema piramidale, come in Figura 5.4. Spostandosi lungo l'asse verticale della piramide si constatano le principali caratteristiche e differenze tra i servizi: muovendosi dal basso verso l'alto, per esempio, aumenta la virtualizzazione dell'infrastruttura ma diminuisce il controllo che si ha dell'applicazione.

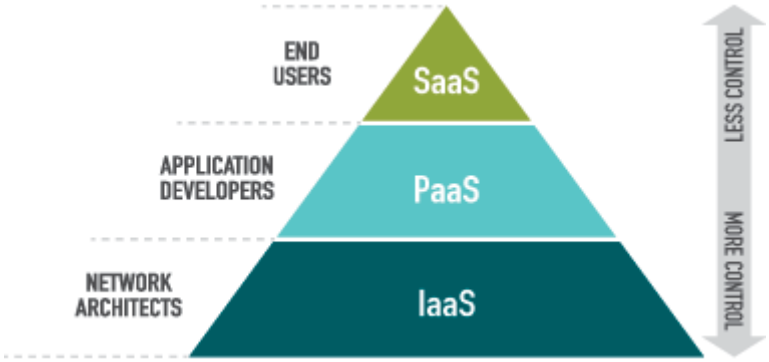

Figura 5.4 - Rappresentazione piramidale dei servizi IaaS, PaaS e SaaS [13]

Il modello IaaS è considerato la base dei servizi di cloud computing. Fornisce solo un'infrastruttura base, quindi l'utilizzatore dovrà preoccuparsi di configurare e gestire l'ambiente di sviluppo e di installare le applicazioni su di esso.

Il secondo modello, PaaS, fornisce una piattaforma dove installare, avviare, gestire e monitorare le applicazioni. L'utilizzatore non dovrà preoccuparsi quindi dell'infrastruttura sottostante.

Nel modello SaaS si ha una virtualizzazione totale. Solitamente l'utente finale utilizzatore del servizio potrà accedervi direttamente via web browser. Non esiste quindi assolutamente la complessità di gestione del servizio. Un

esempio di servizi rientranti in questa categoria può essere la webmail di Outlook o Google.

In Figura 5.5 sono rappresentati i livelli di virtualizzazione dei tre modelli, che confermano quanto detto in precedenza. Lo schema, in particolare, mette in risalto per ognuno quali aspetti sono gestiti dal produttore di software e quali vengono demandati al fornitore di servizi cloud.

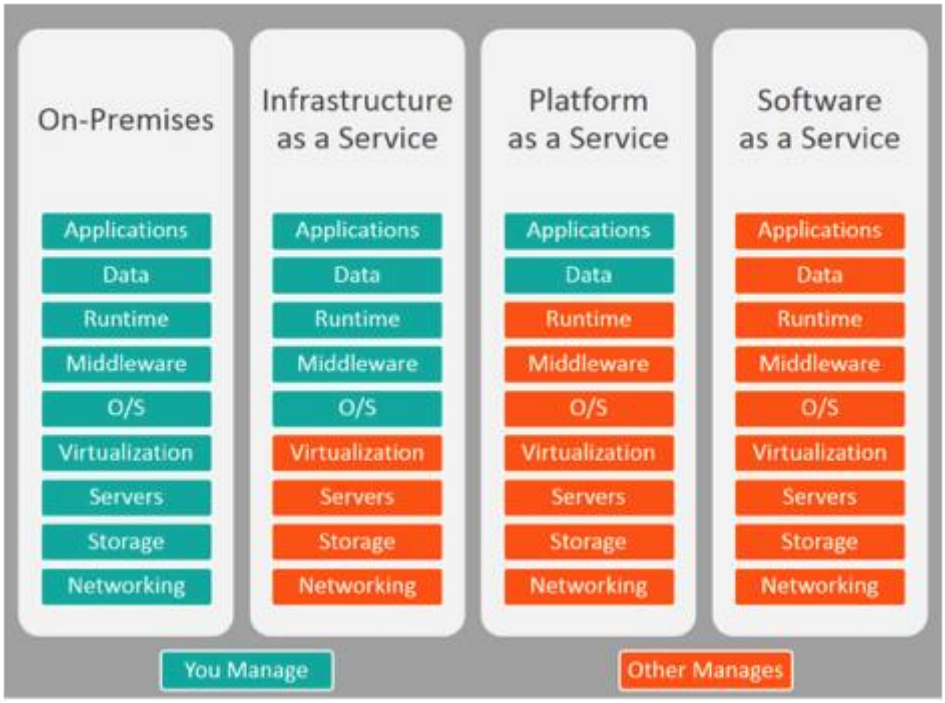

Figura 5.5 - Rappresentazione dei compiti demandati ai fornitori di servizi cloud [14]

Quale delle tre infrastrutture è più opportuno usare per una architettura a microservizi? La virtualizzazione totale del SaaS non si sposa adeguatamente con un'architettura di questo tipo, mentre una IaaS lascia troppi aspetti a carico dello sviluppatore rispetto ad una PaaS. Quest'ultima ad esempio fornisce meccanismi per scalare automaticamente ed una gestione più semplificata dei servizi in fase di esecuzione. Poiché ritengo che una PaaS sia la più calzante per il mio studio, vale la pena fare un ulteriore approfondimento sui possibili modelli e architetture di questa infrastruttura.

6 PLATFORM AS A SERVICE

6.1 ARCHITETTURA

In una Platform as a Service, ogni microservizio è identificato da un endpoint pubblico associato ad esso *(es. https://microservizio1).* Ciascuno di essi esporrà poi dei metodi *(es. https://microservizio1/getCreditoTelefonico)*, che permetteranno di svolgere determinate funzioni o accedere a determinate risorse.

Possono esistere diversi modalità per strutturare l'architettura di una PaaS [15]. In un modello architetturale semplice, rappresentato in Figura 6.1, le applicazioni che necessitano di una determinata funzione invocano direttamente un microservizio installato nella PaaS, mediante una chiamata sull'endpoint messo a disposizione dallo stesso.

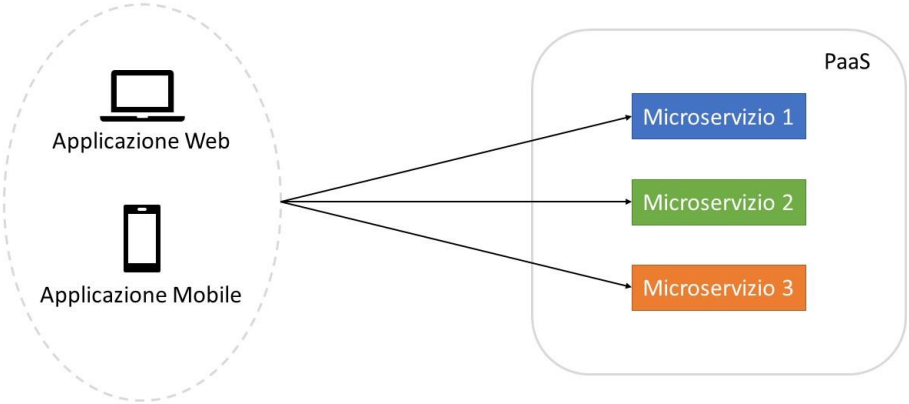

Figura 6.1 - Architettura con chiamata diretta dei microservizi

Questo tipo di architettura introduce una serie di problemi: in primo luogo non si tratta di una soluzione flessibile. Basti pensare che in caso di aggiunta, scissione, unione o modifica in genere di microservizi aggiornare tutti gli endpoint all'interno del client può essere molto dispendioso.

In secondo luogo, un servizio potrebbe utilizzare un protocollo non riconosciuto dall'applicazione, quindi potrebbero verificarsi problemi di comunicazione tra le due parti.

Potrebbero inoltre sorgere problemi di sicurezza, in quanto in questa architettura gli endpoint dei microservizi devono essere richiamabili dall'esterno, rendendoli a tutti gli effetti pubblici. Ciò comporta che ogni microservizio debba prevedere una fase di autenticazione.

Infine, ma non meno importante, questa soluzione probabilmente non si rivelerebbe ottimale da un punto di vista prestazionale se il client dovesse richiamare molti servizi nello stesso momento, in quanto il traffico dovrebbe essere gestito su tutte le terminazioni.

Per risolvere i problemi di cui sopra, viene introdotto un "orchestratore", ossia un Api Gateway.

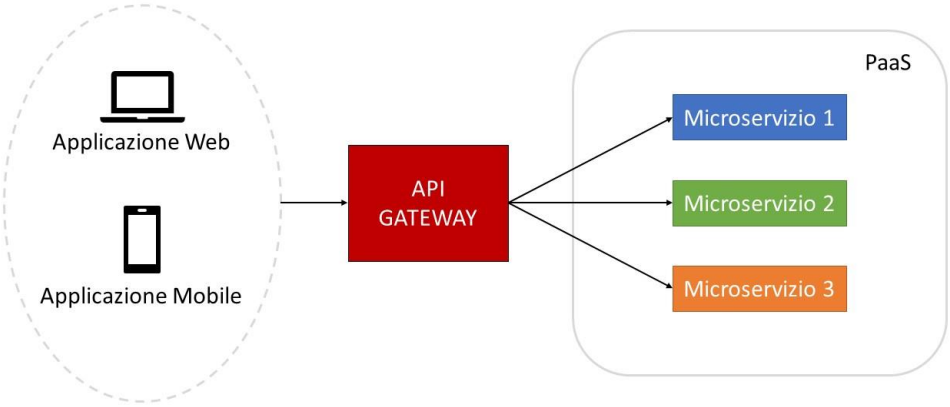

Figura 6.2 - Architettura con un API Gateway

Un Api Gateway è un server posizionato tra le applicazioni e la PaaS che espone un punto d'accesso per le richieste provenienti dalle applicazioni, mettendo a disposizione degli endpoint (es. https://apigateway/getCreditoTelefonico). Si occupa di inoltrare le richieste al microservizio opportuno, preoccupandosi di comunicare con il protocollo adeguato. Può essere usato per chiamare più microservizi, aggregando i risultati fornendoli al client come risposta unica e semplificando le comunicazioni tra applicazione e microservizi.

Un’architettura di questo tipo (Figura 6.2) permette inoltre di non esporre pubblicamente i microservizi, rendendoli contattabili solo dal gateway. A questo proposito, è possibile spostare sul gateway la fase di autenticazione: in questo modo, in caso di richieste su più microservizi, l’autenticazione avviene comunque una sola volta, al contrario dello schema precedente.

Infine, il gateway può occuparsi anche di monitoraggio dei log, o di smistamento del traffico, nel caso di architetture “particolari”. Per fare un esempio, pensiamo ad un’architettura di questo tipo, in cui si vogliono creare gruppi di microservizi con differenti livelli di sicurezza:

- Un gruppo di microservizi ad accesso pubblico, quindi il gateway provvederà solo ad inoltrare le richieste e fornire i risultati.
- Un gruppo ad accesso privato, quindi il gateway prima di processare le richieste dovrà autenticare il client.
- Un gruppo di servizi dedicato a un limitato gruppo di utenti, quindi oltre che autenticare il gateway dovrà applicare determinate regole di controllo degli accessi.

In generale questo tipo di architettura è sicuramente più efficiente della prima, sebbene comporti degli svantaggi: il primo è l’inserimento di un nuovo server nell’architettura, che introduce complessità di installazione e gestione. Il secondo è che in caso di traffico elevato il gateway potrebbe diventare un collo di bottiglia, rallentando l’intera architettura. È infatti importante effettuare un’attenta configurazione, per cercare di limitare al massimo questo fenomeno.

Tenendo presente questi potenziali problemi, questa architettura è sicuramente valida per situazioni di piccola-media grandezza. Ma al crescere dell’applicazione ovviamente segue una crescita della complessità del gateway, il quale somiglierà sempre di più ad un’applicazione monolitica.

Per evitare questo fenomeno, in grandi contesti si possono utilizzare più macchine che fungano da Api Gateway. Un esempio di implementazione può essere visto in Figura 6.3, nel quale ad esempio si utilizza un gateway per l’applicativo web ed uno per l’applicazione per smartphone. Ovviamente in questo modo la complessità viene divisa su due server, rendendo ognuno di essi più gestibile.

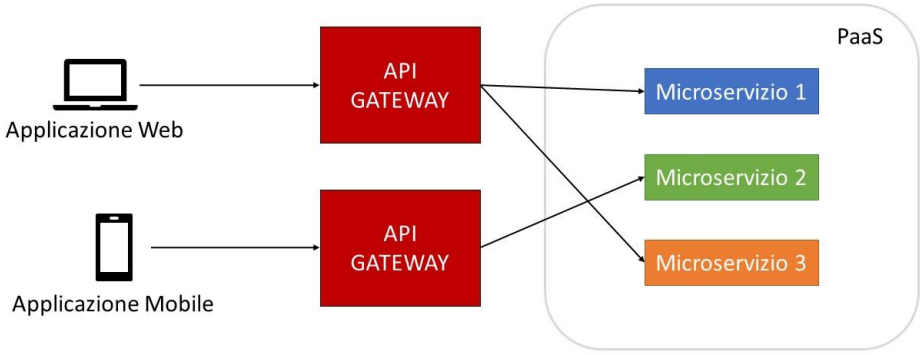

Figura 6.3 - Architettura con più API Gateway

6.2 MODELLI

Esistono fondamentalmente tre modelli di implementazione di una PaaS: locale (detta anche cloud privato), cloud pubblico e ibrido.

Questi modelli differiscono principalmente per i servizi offerti dal Service Provider (o Cloud Provider, il fornitore dei servizi cloud) e conseguentemente per l'aspetto economico.

Il modello di PaaS locale è ottenuto virtualizzando servizi e procedure basate sul cloud su delle infrastrutture IT già esistenti, solitamente di proprietà dell'azienda che vuole sviluppare installare i servizi sulla PaaS. In questa soluzione resta a carico dell'azienda la manutenzione dell'infrastruttura ed il possesso dei dati. Il service provider invece fornisce i servizi da installare per virtualizzare la PaaS sulle macchine esistenti. Questa soluzione non fornisce tutti i vantaggi derivati dall'utilizzo del cloud, ma, prendendo in considerazione il lato economico, è la più conveniente.

Nel modello di PaaS basata interamente sul cloud (Cloud Pubblico), il service Provider provvede a fornire l'intera infrastruttura, dalle macchine agli applicativi, per gestire ed utilizzare la PaaS, accendendovi in remoto. Come facilmente intuibile, questa è la soluzione più onerosa dal punto di vista economico, ma priva l'azienda di tutte le complessità legate alla manutenzione dell'infrastruttura.

Il terzo modello, quello ibrido, garantisce una gestione semplificata dell'infrastruttura, in quanto rende possibile il demandare alcuni servizi al fornitore dei servizi cloud. Si tratta di un modello che può variare molto da situazione a situazione, ma solitamente è utilizzato per utilizzare l'infrastruttura interna (conservando ad esempio l'utilizzo dei data center) integrandola e potenziandola con dei servizi offerti dal provider. L'implementazione può richiedere particolare attenzione, soprattutto sull'integrazione tra l'infrastruttura esistente ed il cloud. Questo modello è un ibrido anche dal punto di vista economico, ed ovviamente l'onere varia in funzione della quantità dei servizi richiesti al provider.

7 MICROSERVIZI: METODOLOGIA DI SVILUPPO

Prima di procedere con l'esposizione delle scelte effettuate per strutturare la soluzione proposta vorrei fare un ulteriore focus sullo sviluppo di microservizi, illustrando le principali metodologie ed i principali pattern architetturali con cui mi sono confrontato o che reputo indispensabili per una soluzione ottimale.

Nel contesto dello sviluppo di software distribuito, installato nel cloud e rilasciato sotto forma di servizi, è bene seguire una serie di "regole" durante lo sviluppo, al fine di implementare nell'applicazione i corretti livelli di sicurezza, efficienza e scalabilità. A questo proposito, il co-fondatore di Heroku (prodotto che offre dei servizi PaaS) Adam Wiggins sintetizzò le "best practices" in una serie di concetti base, sviluppando un approccio ottimale per rilasciare servizi sulla PaaS Heroku. Tuttavia, si notò che questo approccio, consistente in 12 "fattori" fondamentali, poteva essere esteso anche a chi utilizzava altri modelli di PaaS o a chi sviluppava SaaS, in quanto rappresentava semplicemente le migliori pratiche da implementare nello sviluppo di applicazioni cloud. Fu così che per gli sviluppatori DevOps la cosiddetta metodologia "12 Factor App" divenne quasi uno standard da seguire.

7.1 TWELVE FACTOR APP

Twelve Factor App abbraccia totalmente le metodologie agili descritte finora. Le applicazioni scritte seguendo questo approccio ottengono direttamente tutti i principali vantaggi del mondo DevOps, tra cui dichiarazione chiara di requisiti, configurazioni e funzionalità, al fine di favorire l'ingresso di nuovi sviluppatori, adattabilità all'installazione su infrastrutture cloud, possibilità di implementare il concetto di Continuous Deployment.

Twelve-factor App può in generale essere applicata durante un qualsiasi tipo di sviluppo software, ed è indipendente dal linguaggio di programmazione scelto o da eventuali servizi esterni (ad esempio database). [16]

I dodici fattori su cui basare lo sviluppo sono i seguenti:

1. Codebase

   Con il termine codebase si intende l'insieme del codice sorgente dell'applicazione. È fondamentale che sia presente un'unica codebase, ossia un unico repository (letteralmente "deposito"). All'interno del repository è possibile rilasciare vari deployments, ossia diverse istanze dell'applicazione: un deployment può ad esempio essere un nuovo rilascio effettuato in seguito ad un aggiornamento dell'applicazione, o una singola versione posseduta in locale da un singolo sviluppatore. Nell'ottica di poter gestire con ordine tutte le istanze all'interno della codebase è fondamentale utilizzare un sistema per il versionamento di ogni istanza (ad esempio GIT [17]).

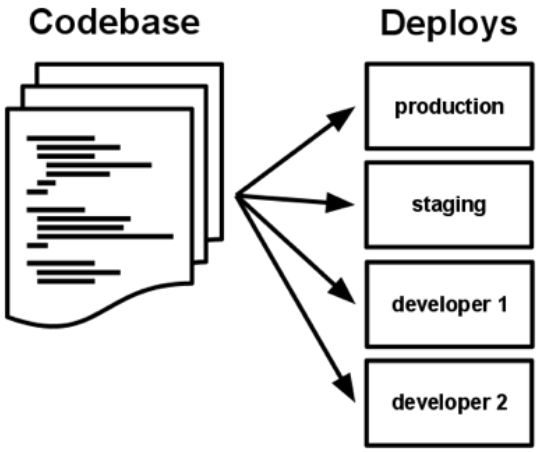

Figura 7.1 - Rappresentazione di una unica codebase in cui sono presenti molteplici deployments [18]

2. Dipendenze

   La gestione delle dipendenze, quindi l'utilizzo di eventuali librerie esterne, deve avvenire in maniera molto precisa. Esse devono essere isolate e dichiarate: isolate perché non devono essere contenute nella codebase, ma devono essere recuperate in real time da repository esterni; a tal proposito, devono essere dichiarate in una sorta di "contratto" (file manifest) con l'applicazione. Ciò consente ad ogni sviluppatore di avere una visione chiara e precisa di tutte le dipendenze.

3. Configurazione

   La “configurazione” di un servizio consiste in tutto ciò che può cambiare tra i vari deployments. Possono essere elementi di configurazione le credenziali per accesso a servizi esterni, i parametri di connessione al db o generiche variabili utilizzate all’interno dell’applicazione. È molto importante dichiarare questi parametri come “variabili d’ambiente”, e non come costanti all’interno del codice, in quanto quest’ultimo può non cambiare tra diversi deployments, al contrario delle configurazioni.

4. Backing Service

   Per Backing Service si intende un servizio di terze parti utilizzato dall’applicazione, come ad esempio un database, un server di autenticazione o un servizio di code/messaging.

   Nel Twelve Factor App un servizio locale ed un servizio di terze parti vengono considerati e trattati in ugual modo come risorse. I parametri di connessione a queste risorse devono essere dichiarati nel file di configurazione, in modo da poter passare da una risorsa ad un'altra equivalente senza apporre modifiche al codice.

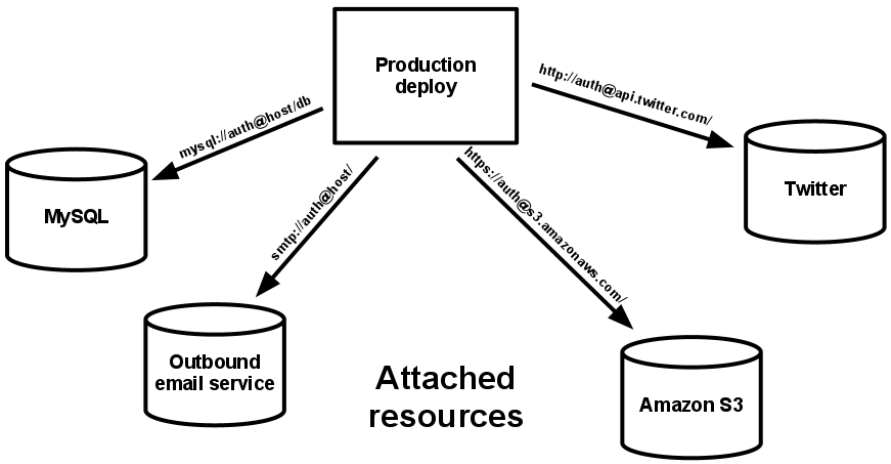

Figura 7.2 - Rappresentazione di servizi in background [19]

Come nell’esempio riportato in Figura 7.2, modificando ad esempio la URL di connessione al servizio di autenticazione e i relativi

parametri di configurazione sarebbe possibile passare da un'autenticazione su Amazon ad una su Twitter.
Ciò garantisce una maggiore tolleranza ai guasti in quanto, in caso di malfunzionamento di una risorsa, è possibile "scollegarla" e utilizzarne un'altra utile al nostro scopo.

5. Build, release, esecuzione
   Come si ottiene un deployment? La risposta è nel quinto fattore, cioè mediante tre fasi: build, release ed esecuzione. La prima fase fa una "costruisce" (build) il codice presente nella codebase, creando un eseguibile con all'interno le dipendenze associate. La seconda fase associa alla build tutte le configurazioni specifiche di quel rilascio, creando una release. A questo punto l'applicazione può essere lanciata: si arriva alla terza fase, la fase di esecuzione.

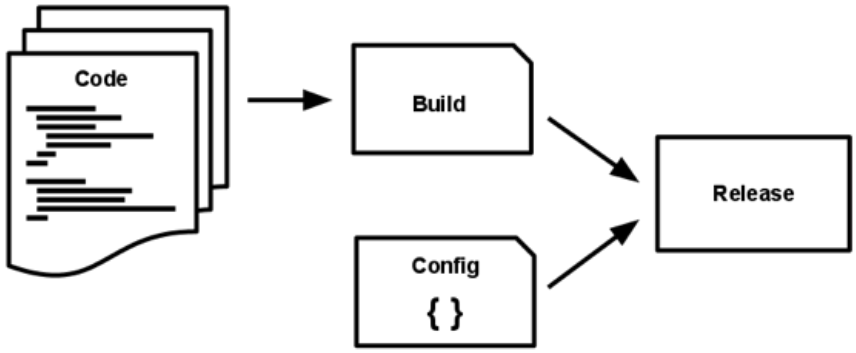

Figura 7.3 - Rappresentazione delle operazioni svolte per ottenere una release [20]

La metodologia twelve factor prevede una netta separazione tra queste tre fasi, che devono inoltre verificarsi necessariamente in questo ordine. Non sono previste quindi ad esempio modifiche del codice nella fase di esecuzione (modifiche a runtime): infatti in caso di modifiche si dovrà rieffettuare la fase di build e ripartire con i passi descritti.

6. Processi
   L'applicazione quando è in stato di esecuzione esegue uno o più processi per servire le richieste in ingresso. È importante che ogni processo non conservi variabili temporanee o non utilizzi meccanismi di cache, e che le operazioni eseguite dal processo siano quanto più

atomiche possibile. “I processi Twelve Factor sono inoltre stateless (senza stato) e share-nothing (non condividono nulla con altri processi)” [21]. Ogni dato persistente deve essere memorizzato in un backing service.

7. Binding delle Porte

   Al contrario di molte applicazioni che per essere in stato di esecuzione hanno bisogno di un “contenitore” (tipicamente un server web), le applicazioni Twelve Factor sono autocontenute e non necessitano di contenitori per esporre i propri servizi. Esse effettuano una assegnazione delle porte (binding) specifica per determinati servizi, fornendo una URL completa sulla quale esporre il servizio stesso. In questo modo possono fungere da backing service per altre applicazioni.

8. Concorrenza

   Il concetto di concorrenza riprende il fattore che riguardava i processi. Sappiamo che un’applicazione in stato di esecuzione è composta da uno o più processi. È possibile implementare la concorrenza istanziando più processi. Occorre innanzitutto fare una divisione logica, assegnando ogni processo ad un tipo di lavoro, per implementare una sorta di scalabilità orizzontale. In seguito, a seconda del carico di lavoro, ogni tipo di processo può essere istanziato nuovamente, aumentando il numero di processi attivi e scalando verticalmente l’applicazione.

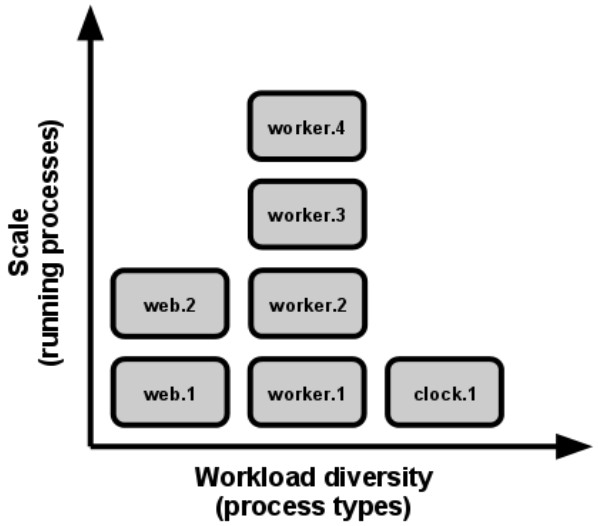

Figura 7.4 – Scalabilità e concorrenza [22]

9. Rilasciabilità

   La rilasciabilità è una qualità che deve essere posseduta dai processi dell'applicazione. Con il termine rilasciabilità si intende che essi potranno essere avviati o stoppati al bisogno su richiesta, in maniera "agile", mirando a velocizzare quanto più possibile le fasi di deploy e di avvio dell'applicazione, ed a rendere quanto più possibile la stessa tollerante ad errori improvvisi.

10. Parità tra Sviluppo e Produzione

    Nello sviluppo DevOps in generale esistono due ambienti nettamente separati in cui viene rilasciato il software, ossia l'ambiente di sviluppo e quello di produzione. Solitamente esistono delle sostanziali differenze tra questi due ambienti riguardanti le tempistiche di rilascio, gli stack di tecnologie utilizzate e gli sviluppatori coinvolti.

    Il principio di questo fattore è quello di limitare al massimo queste differenze.

11. Log

    Nelle classiche applicazioni solitamente i log vengono salvati in un apposito file su disco. In un'applicazione Twelve Factor è necessario trattare i log come uno stream di eventi sequenziali. Durante lo sviluppo in locale questo stream può essere restituito sullo standard

output, così da renderlo di rapida consultazione. Mentre l'applicazione è in esecuzione, invece, è bene indirizzare lo stream verso sistemi appositi per la gestione dei log, che permetteranno la consultazione degli stessi, di statistiche ed errori in tempo reale.

12. Processi di Amministrazione

    L'ultimo fattore riguarda i cosiddetti processi di amministratore: quei processi che vengono effettuati saltuariamente per la gestione o la manutenzione dell'applicazione (come la migrazione di un database o l'esecuzione di script all'interno dell'ambiente). Secondo questo approccio questi processi dovrebbero essere eseguiti prima dell'avvio dell'applicazione. Per poter invece intervenire in tempo reale anche quando l'applicazione è in esecuzione, occorrerebbe lavorare su una nuova release, che condivida lo stesso codice e le stesse configurazioni della versione attiva. Dopo gli interventi ed una fase di test, si procederà con l'avvio della release "aggiornata", che sostituirà la precedente.

7.2 CARATTERISTICHE ARCHITETTURALI DI UN MICROSERVIZIO

Consideriamo ora l'aspetto più "architetturale". Anche in questo caso esistono dei principi di design, utili per strutturare un microservizio in maniera ottimale. Queste "regole" sono brevemente descritte sotto.

- *Service Contract*: poiché ogni microservizio interagisce con altri client deve fornire, oltre all'endpoint, anche un "contratto" che lo descriva, che contenga tutte le specifiche dello stesso.
- *Service Loose Coupling*: i microservizi devono essere disaccoppiati e indipendenti, al fine di ridurre al minimo le dipendenze esterne.
- *Service Abstraction*: il Service Contract deve contenere solo informazioni essenziali ed effettivamente utili alla funzionalità del servizio.
- *Service Reusability*: i microservizi devono poter essere richiamati anche da altre parti del software, e possono quindi essere riutilizzati.
- *Service Autonomy*: i microservizi devono risultare indipendenti dall'ambiente di esecuzione.
- *Service Composability*: ogni servizio deve essere "componibile". Può essere composto da più servizi o può esso stesso fare parte di uno.

- *Service Statelessness*: al fine di ridurre le risorse utilizzate, nessun servizio al termine della sua funzione può conservare dei dati in memoria.
- *Service Discoverability*: i microservizi avvisano l'ambiente della loro presenza e disponibilità. Allo stesso modo quando diventano indisponibili notificano il cambiamento di stato, rimuovendosi dalla lista delle risorse raggiungibili

7.3 PRINCIPALI DESIGN PATTERN – CIRCUIT BREAKING

Generalmente, ogni servizio per interagire con altri effettua delle richieste remote sulle URL appositamente esposte. Consideriamo un caso d'esempio: un servizio A, prima di proseguire con il suo flusso di operazioni, richiama più volte un servizio B. Se il servizio B non fosse in grado di rispondere entro un tempo massimo, ad esempio per un sovraccarico o per un malfunzionamento, A dovrebbe per ogni chiamata attendere lo scadere di un timeout prima di recepire l'errore, che si verificherebbe in ogni caso per ogni chiamata effettuata.

Il pattern Circuit Breaking permette di rilevare e gestire in maniera efficiente tali errori, realizzando via software [23] l'equivalente di un interruttore in un circuito elettrico, per interrompere il flusso di un servizio.

Un Circuit Breaker è strutturato come un proxy che controlla il flusso verso un endpoint di un servizio. Nel caso in cui su quest'ultimo si verifichino un numero di fallimenti superiore ad una soglia prestabilita, il proxy "apre" il circuito, rifiutando le richieste verso quel servizio e dirottandole nella maggior parte dei casi verso un bilanciatore, il quale instraderà la richiesta verso un altro container. Il circuito rimarrà aperto per un certo intervallo di tempo (scelto in fase di configurazione), dopodichè passerà in una fase in cui viene definito "semiaperto". A quel punto il proxy deciderà, a seguito dell'esito di una successiva richiesta, di aprire o chiudere il circuito

La Figura 7.5 rappresenta un diagramma di transizione degli stati:

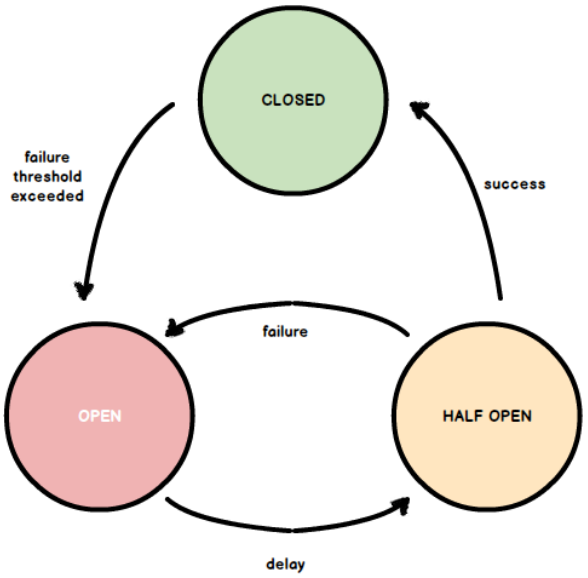

Figura 7.5 – Diagramma di transizione degli stati di un Circuit Breaker [24]

7.4 PRINCIPALI DESIGN PATTERN – GREEN/BLUE DEPLOYMENT

Uno dei rischi più grandi nel rilascio software è quello di rilasciare in ambiente di produzione una versione dell'applicativo contenente errori o bug.

Il Pattern di rilascio Green/Blue propone un metodo per limitare al massimo questo rischio. Consiste nel creare due ambienti di produzione identici, entrambi connessi alle stesse risorse e ai medesimi database. Solitamente vengono chiamati Green e Blue e sono destinati uno ad uso interno ed uno ad uso effettivo degli utenti.

Se, come nell'esempio rappresentato in Figura 7.6, assumessimo che in fase iniziale l'ambiente di effettivo di produzione sia il Green, il prossimo rilascio avverrà in prima fase in ambiente Blue. Su quest'ultimo si potranno effettuare tutti i test necessari al fine di assicurarsi che il rilascio non introduca errori. In caso di esito positivo, l'ambiente Blue diventerà l'ambiente di produzione principale, ed il Green verrà aggiornato con lo stesso rilascio, diventando però l'ambiente interno in cui verrà testato il prossimo.

Si ha quindi una continua alternanza tra Green e Blue, che diventeranno a loro volta l'ambiente di produzione principale.

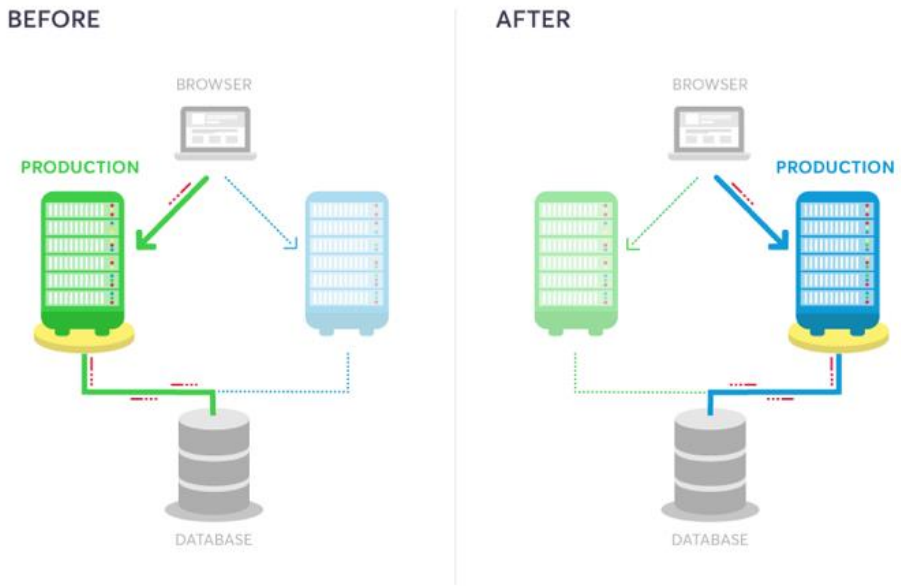

Figura 7.6 – Rappresentazione del Green/Blue Deployment pattern [25]

7.5 PRINCIPALI DESIGN PATTERN – CANARY DEPLOYMENT

Un altro pattern che nasce per limitare i rischi causati da nuovi rilasci è quello del Canary Deployment. In maniera similare al Green/Blue Deployment, si basa sul fatto di non rilasciare le nuove versioni degli applicativi direttamente al posto del software esistente.

Seguendo questo pattern in fase di nuovo rilascio si dovrebbe creare un ambiente isolato contenente la nuova versione, in cui gli utenti non hanno inizialmente accesso ed è possibile testare la nuova release del software.

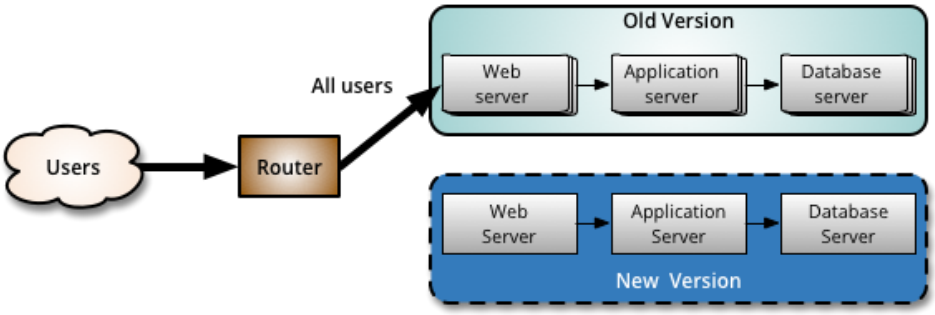

Figura 7.7 – Canary Deployment, prima fase [26]

In seguito all'esito positivo dei test della nuova versione, sarà possibile renderla disponibile ad un limitato numero di utenti, che si comporteranno come beta-tester.

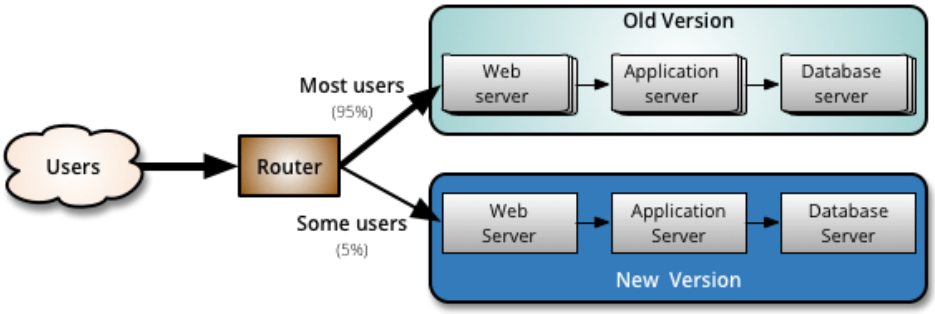

Figura 7.8 – Canary Deployment, seconda fase [27]

Questo test potrebbe evidenziare ulteriori problemi o errori, per cui si è ancora in tempo utile per apporre le opportune correzioni. Quando la nuova versione risulta sufficientemente robusta, si può reindirizzare su di essa la totalità degli utenti, provvedendo ad eliminare la vecchia versione.

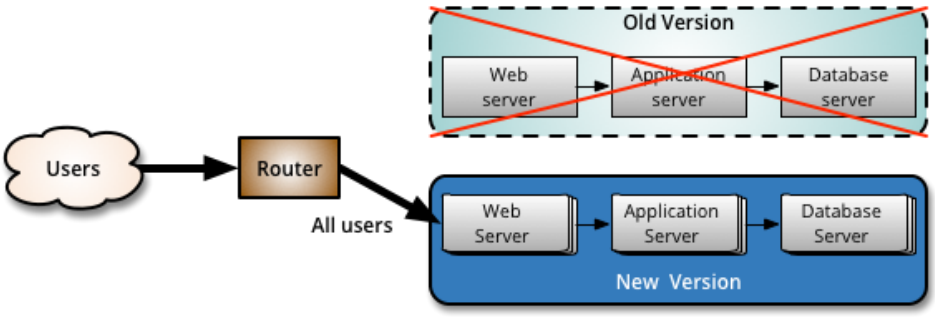

Figura 7.9 – Canary Deployment, terza fase [28]

8 AMBIENTI DI SVILUPPO E TECNOLOGIE PROPOSTE

Dopo aver trattato i principali elementi e le principali tecniche, illustrerò nel dettaglio lo stack di tecnologie utilizzato per implementare la mia soluzione.

8.1 LA PAAS E REDHAT OPENSHIFT

Per quanto riguarda l'infrastruttura ho optato per l'utilizzo di una PaaS locale, disponendo già di una macchina che potesse ospitare il sistema. Ho inoltre tenuto molto in considerazione l'aspetto economico, che in molte situazioni risulta essere il primo fattore di scelta.

Tra i vari prodotti commerciali, quello che mi è sembrato più in linea con le mie esigenze è stato RedHat Openshift, il quale offre due soluzioni di PaaS: una soluzione cloud nella versione Openshift Online ed una locale nella versione Container Platform.

RedHat Openshift è uno dei più diffusi servizi offerti dal mercato per l'implementazione di una PaaS, ed offre un'ottima gestione delle fasi di build, deploy e manutenzione di applicazioni in container.

"Con OpenShift è possibile implementare servizi distribuiti, avanzati, scalabili, installati in dei container, i quali combinano le applicazioni insieme ai file da cui dipendono, rendendo uniformi gli ambienti di sviluppo e di produzione, semplificando la distribuzione delle applicazioni." [29] Una caratteristica importante di Openshift è quella di supportare diversi linguaggi, regalando una flessibilità aggiuntiva allo sviluppatore.

8.2 MICROSERVIZI

La PaaS di Openshift supporta tre tra i principali frameworks o piattaforme utilizzati nello sviluppo di microservizi: Spring, Node.js e .NET. Concentrandomi sui primi due, in quanto open source, ho fatto un confronto per scegliere come implementare il servizio da me proposto.

8.2.1 Spring Boot

Spring è uno dei framework utilizzati per sviluppare applicazioni JAVA. Nasce nel 2002 e viene rilasciato per la prima volta nel 2003 con la versione 1.0 in risposta alla complessità delle prime specifiche di J2EE [30]. È un framework open source di cui sono stati effettuati diversi aggiornamenti, l'ultimo dei quali nel 2007, con la versione 5.0.

Nel corso degli anni, per rimanere coerente alle nuove tecnologie ed ai nuovi modelli di programmazione, Spring è stato integrato con diversi progetti. Uno di questi è Spring Boot, il quale permette di sviluppare applicazioni secondo i principi DevOps e adatti all'utilizzo in cloud. Spring Boot semplifica e velocizza la creazione di applicazioni indipendenti (nel nostro caso microservizi), limitando al massimo le fasi di configurazione, come la gestione delle dipendenze, le connessioni con i database e l'esposizione di api, gestendole con poche righe di codice o annotazioni.

Ad esempio, in merito alla gestione delle dipendenze, Spring Boot offre un servizio che permette di inizializzare in maniera automatica un progetto. Selezionando all'interno di un form le funzionalità che vorremo sfruttare all'interno dell'applicazione, saremo in grado di scaricare un progetto che contenga le dipendenze già configurate.

Se in un certo momento nel nostro progetto venisse aggiunta una dipendenza relativa ad un database MySQL, il framework assumerebbe che probabilmente avremo necessità di utilizzarlo. Per questo motivo si occuperebbe dell'autoconfigurazione dell'applicazione per l'accesso ad un database MySQL.

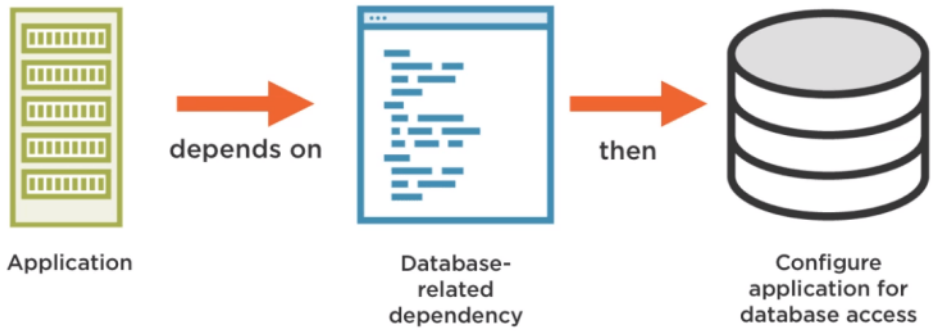

Figura 8.1 – Esempio di autoconfigurazione di Spring Boot [31]

Per usufruire dell'autoconfigurazione di Spring Boot, dovremo solo aggiungere in testa alla dichiarazione della classe principale l'annotazione *@EnableAutoConfiguration*.

Un'altra caratteristica importante di un'applicazione Spring Boot è quella di essere Stand-Alone (indipendente da ogni altro sistema). Solitamente, per avviare un'applicazione web scritta in JAVA ci sono diversi step da eseguire: effettuare il package dell'applicazione, scegliere e scaricare un web server, configurarlo, installare l'applicazione ed infine avviare il server.

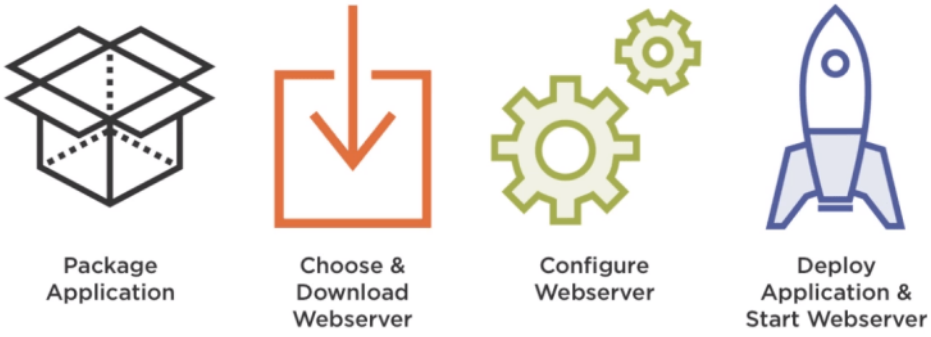

Figura 8.2 - Processo per avviare un'applicazione web scritta in JAVA [32]

Tramite Spring Boot, che si preoccupa di avviare e configurare autonomamente un web server, è possibile invece rendere disponibili i nostri servizi in soli due passaggi: effettuare il package dell'applicazione ed avviarla.

Le funzionalità offerte da Spring Boot sono in continuo aumento, dato che il progetto è ancora in rapida evoluzione e viene costantemente aggiornato.

8.2.2 Node.js

Node.js è un framework che permette di eseguire del codice lato server in linguaggio JavaScript. È una piattaforma Open Source nata nel 2009, essa stessa realizzata in gran parte in JavaScript, in costante sviluppo ed aggiornamento. Nonostante il linguaggio JavaScript fosse un linguaggio tipicamente usato per lo sviluppo di applicazioni lato client, grazie a Node.js è possibile implementare il paradigma "JavaScript everywhere",

permettendo di utilizzare un solo linguaggio di programmazione durante lo sviluppo di un'applicazione web.

Secondo il "Developer Survey", un sondaggio effettuato ogni anno da Stack Overflow, nota piattaforma di community per sviluppatori, JavaScript è il linguaggio più popolare, o in generale il più comunemente utilizzato, tra gli informatici [33].

Nel Grafico 8.3 è rappresentato l'esito del sondaggio relativo alla fine del 2017.

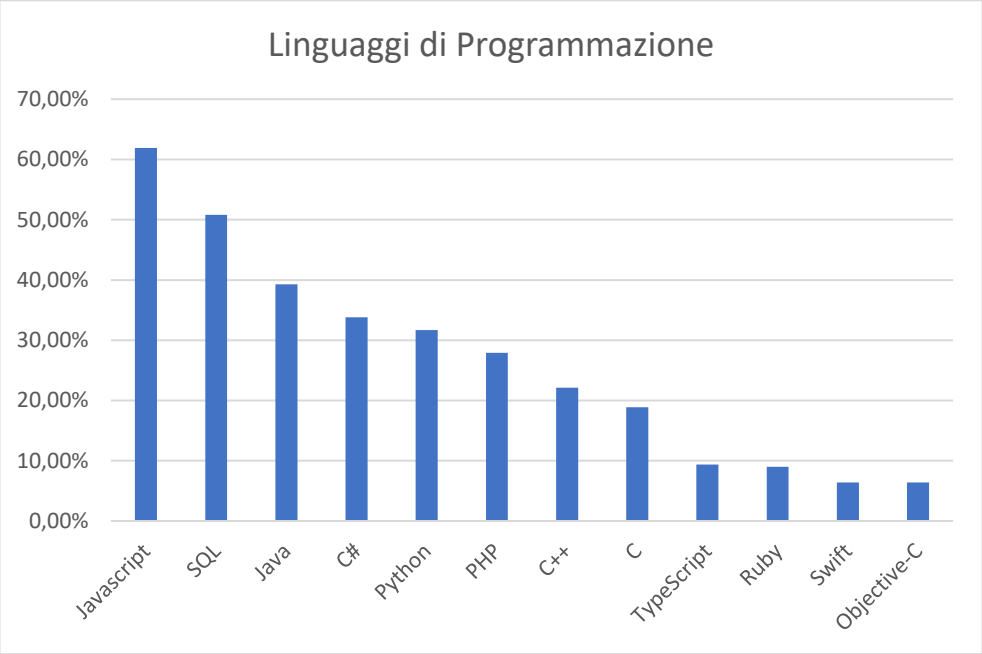

Grafico 8.3 – Percentuale di utilizzo dei principali linguaggi di programmazione

Tutto ciò può rappresentare sicuramente un primo vantaggio, in quanto l'utilizzo di un unico linguaggio, per di più molto popolare, attrae molti sviluppatori verso l'utilizzo di Node.js.

Un'altra caratteristica positiva di Node.js è la velocita di esecuzione, data in parte dal motore "Google V8 engine" [34], che si occupa della compilazione del codice, ottimizzato per la velocità del Javascript, ed in parte dal fatto che Node.js è una piattaforma "event-driven": utilizza delle chiamate asincrone non bloccanti, eseguite in background, mostrando all'utente finale maggiore reattività.

Node.js supporta numerosi frameworks per lo sviluppo di microservizi, fra cui Seneca, i quali riescono ad implementare al meglio la metodologia DevOps.

8.2.3 Confronto

Dopo sviluppato ed installato dei semplici microservizi di test, sia con Spring che con Node.js, ho notato poche lievi differenze.

- Node.js si è rivelato leggermente più veloce e reattivo di Spring, sia nell'avvio del servizio, ma anche in fase di risposta alle richieste.
- Node.js gestisce meglio le interazioni con database non relazionali (come Mongo DB), mentre Spring, sfruttando le Java Persistent API (JPA) [35], risulta migliore nel comunicare con database relazionali (i classici MySQL e Oracle).
- Spring sembra offrire più pattern per la sicurezza, offrendo cookie, autenticazione classica, token, Kerberos e OpenAM, non tutti disponibili per Node.js.

Infine, un aspetto che ritengo molto importante per uno sviluppatore, è il "trend" delle due tecnologie. Sapere quanto siano usate, diffuse e quanto vengono citate sul web è secondo me fondamentale nel lavoro di uno sviluppatore, che potrà avere più facilmente accesso a documentazione, blog e community dove saranno presenti consigli di implementazione o risoluzioni di errori comuni. Ho quindi fatto un'analisi su Google Trends, servizio offerto da Google che rappresenta tramite un grafico la quantità di ricerche tra due termini, nel nostro caso Spring Boot e Node js.

Nel trend degli ultimi 12 mesi ho constatato come anche il volume delle ricerche effettuate su Google per questi due termini sia simile, nonostante negli ultimi mesi anno ci sia stato un passo in avanti di Spring rispetto a Node.js.

Sebbene a livello prestazionale non abbia riscontrato grosse differenze tra i due ho scelto di utilizzare il framework Spring. Mi ha convinto maggiormente soprattutto la possibilità di JAVA di testare il codice già in compile-time (grazie alla presenza di un compilatore), cosa non possibile per Javascript.

8.3 CONTAINER

Dopo aver affrontato il tema dell'indipendenza dei microservizi, sorge spontanea una domanda. Come renderli davvero autonomi durante il loro ciclo di vita, quando più di essi utilizzano le risorse dello stesso sistema operativo? La risposta può essere introdotta spiegando il concetto di Container Linux.

"I container Linux sono tecnologie che consentono di raccogliere e isolare le applicazioni attraverso un ambiente runtime completo, vale a dire attraverso tutti i file necessari per l'esecuzione". [36] Sono realizzati tramite una sorta di virtualizzazione del sistema operativo, nella quale più container condividono lo stesso kernel.

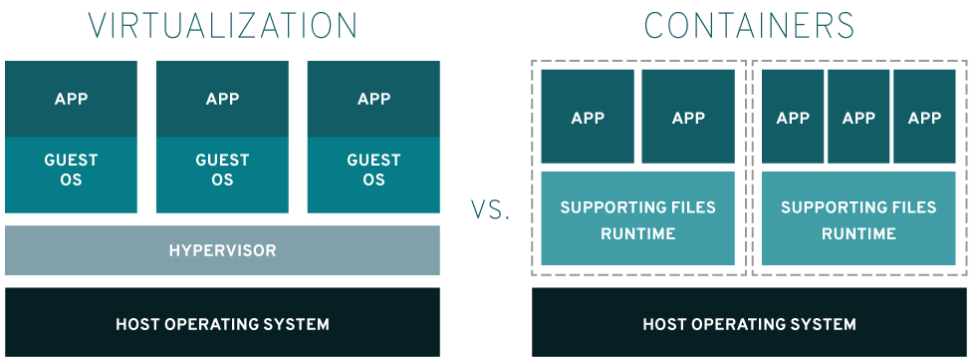

Figura 8.4 – Rappresentazione dell'architettura della virtualizzazione e dei container [37]

I container risultano molto efficienti soprattutto in caso di numerose applicazioni eseguite contemporaneamente, in quanto non sprecano risorse per l'hypervisor (il componente che virtualizza) e per i molteplici sistemi operativi.

8.3.1 Docker

La più conosciuta tecnologia per creare e utilizzare dei container è Docker, un progetto open source che nasce nel 2008. "Docker considera i container come macchine virtuali modulari estremamente leggere, offrendo la flessibilità di creare, distribuire, copiare e spostare i container da un ambiente all'altro". [38] Docker è nato dalla tecnologia LXC, ossia quella dei

container Linux tradizionali, ma differisce da essa in quanto riesce ad isolare i processi, rendendoli indipendenti, con un conseguente aumento della modularità dell'applicazione. Fornisce dei metodi più agili per gestire l'installazione delle immagini nel container, con eventuali rollback, e il versioning delle stesse.

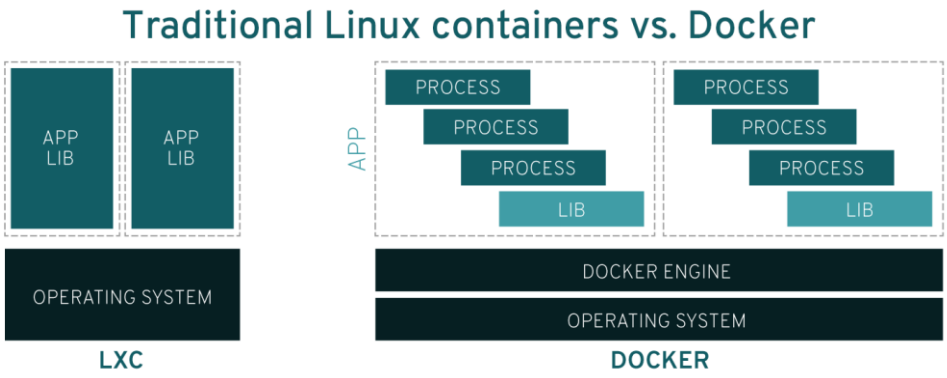

Figura 2.5 - Rappresentazione dell'architettura del container Linux e di Docker [39]

Docker gestisce molto efficacemente i singoli container, ma quando un'applicazione ne contiene molteplici (come nella maggior parte delle situazioni reali), la coordinazione tra essi può diventare molto complessa. Ecco perché al fianco dei container Docker è necessario introdurre la figura di un "orchestratore", che possa gestire al meglio la convivenza tra più container.

8.4 Orchestrazione: Kubernetes

L'orchestrazione comprende tutti gli aspetti legati al ciclo di vita dei container, soprattutto in caso di ambienti dinamici e di dimensioni considerevoli. In particolare, si occupa di installare e rendere disponibili i container, scalarli istanziandoli nuovamente, raggrupparli in cluster, rimuoverli o spostarli, ma anche di gestirli quando sono richiesti dall'esterno, allocando le risorse, preoccupandosi di bilanciare il carico di richieste in entrata e monitorando il loro stato di attività.

La soluzione più diffusa è certamente Kubernetes, sistema di orchestrazione Open Source inizialmente sviluppato da Google, ampiamente compatibile con i container Docker.

Kubernetes raggruppa una serie di macchine fisiche o virtuali, chiamate nodi, e in ognuno di essi possono essere istanziati uno o più POD, l'unità fondamentale nella tecnologia Kubernetes. Un POD è un'istanza in esecuzione di un software, e può essere costituito a sua volta da uno o più container. Può essere creato, clonato, eliminato all'occorrenza, così da garantire la massima scalabilità.

L'elemento che si occupa della creazione o eliminazione dei POD sui singoli nodi è il Controller, i cui comportamenti principali possono essere settati in fase di configurazione.

La piattaforma Openshift scelta integra una versione di Kubernetes che mi ha permesso appunto, come vedremo più avanti, un'allocazione "dinamica" dei vari POD.

9 ARCHITETTURA PROPOSTA

Dopo aver analizzato le principali tecnologie coinvolte, inizio ad illustrare l'applicazione a microservizi proposta come soluzione. In Figura 9.1 è riportato uno schema dell'architettura.

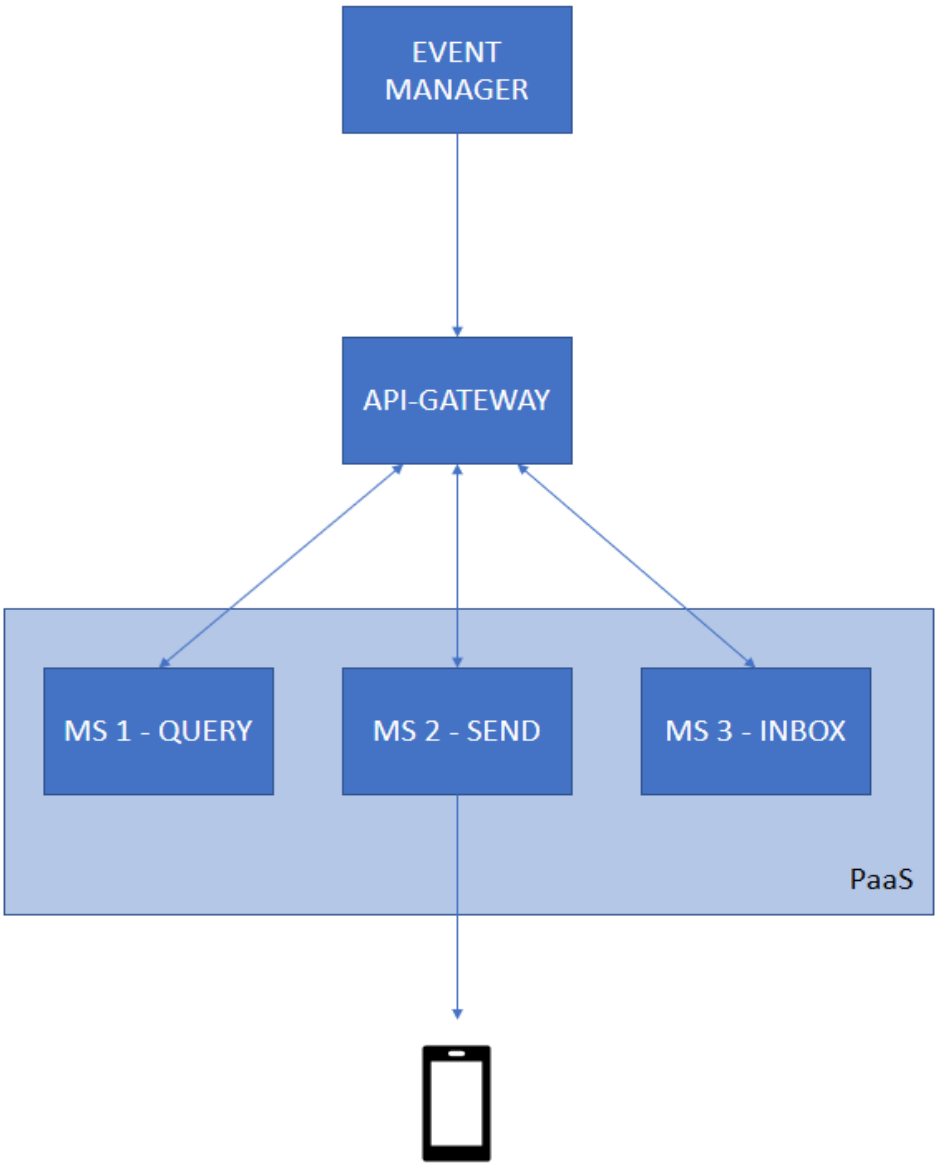

Figura 3.1 – Schema architetturale della soluzione proposta (Elaborazione personale)

In questa architettura l'elemento che scatena il processo di invio della comunicazione è l'Event Manager. Generalizzando, l'ho definito in questo modo in quanto potrebbe trattarsi di una coda di messaggi, di uno schedulatore di eventi o di un server che ha bisogno di inoltrare delle comunicazioni secondo una qualsiasi logica. In ogni caso, quando l'Event Manager avrà necessità di inviare una comunicazione, effettuerà una richiesta ad un Api Gateway, che inizierà a processare la richiesta.

9.1 API GATEWAY

Come illustrato in precedenza l'Api Gateway fungerà da tramite verso la Platform-as-a-Service che ospita i microservizi. In primo luogo autenticherà il client, verificando che esso abbia i corretti requisiti e che la richiesta sia semanticamente corretta. Nel nostro caso la richiesta deve contenere almeno uno o più destinatari, identificati da un parametro (ad esempio dal numero di telefono o dal codice fiscale), ed il testo della comunicazione. Successivamente, l'Api Gateway fornirà questi parametri ai microservizi, ne verificherà l'esito, e restituirà alla fine del processo una risposta all'Event Manager, positiva in caso di un corretto invio del messaggio.

I microservizi all'interno della PaaS sono tre, destinati ai seguenti scopi.

9.2 MICROSERVIZIO 1 – QUERY VERSO IL DATABASE

Il primo microservizio si occupa di "preparare" l'invio della comunicazione. Riceve dal gateway l'identificativo dell'utente destinatario e interroga il database per recuperare il parametro adatto necessario all'invio. Questo parametro potrebbe essere ad esempio l'indirizzo e-mail nel caso di un invio di posta elettronica o un identificativo del dispositivo mobile nel caso di una notifica Push. Restituisce al gateway il valore di questo identificativo.

9.3 MICROSERVIZIO 2 – INVIO DELLA COMUNICAZIONE

Il secondo microservizio si occupa di inviare effettivamente la comunicazione. In un primo momento configurerà il messaggio, recuperando il testo della comunicazione.

In secondo luogo procederà all'invio. Questa fase varierà in maniera considerevole in base al canale scelto. Potrebbe ad esempio richiedere l'invio di una mail ad un Mail Server o occuparsi di comunicare con Google

Firebase e Apple Push Notification per la consegna della notifica sul dispositivo mobile.

Al termine dell'invio del messaggio, che già in questa fase viene quindi recapitato all'utente, risponde al gateway con l'esito della consegna.

9.4 MICROSERVIZIO 3 – SALVATAGGIO DELLA COMUNICAZIONE

Questo microservizio potrebbe non essere necessario per tutti i canali di comunicazione. Si occupa di salvare il messaggio o la notifica recapitati all'utente su un database. Potrebbe essere utile ad esempio in caso di una notifica push, per accedere ad una specifica sezione all'interno dell'app contenente la lista di notifiche ricevute con la possibilità di consultarle nuovamente, mentre potrebbe rivelarsi inutile in caso di invio di comunicazioni tramite e-mail.

10 ANALISI

Dopo aver sviluppato l'architettura a microservizi, è stata effettuata un'analisi circa le prestazioni e l'efficienza dell'architettura in questione. Questa tesi è nata con lo scopo di offrire una soluzione migliore dell'attuale sistema monolitico, quindi è inevitabile fare delle comparazioni tra i due sistemi. Purtroppo non mi è possibile divulgare dati prestazionali reali per motivi di privacy aziendale e per questo motivo ho realizzato un'applicazione monolitica (sviluppata in JAVA) con le stesse funzionalità dell'applicazione a microservizi, in modo da poter confrontare le prestazioni tra le due e riuscire a rendere comunque l'idea delle principali differenze.

Per la comparazione ho effettuato dei test di carico, invocando le API dei due sistemi con un numero sempre maggiore di richieste al secondo e studiando in seguito le principali criticità.

10.1 TEST CON JMETER

Lo strumento utilizzato per i test è Apache JMeter [40], un'applicazione JAVA Open Source in grado di studiare i comportamenti e le prestazioni di API o applicazioni in generale effettuando dei test di carico personalizzati.

JMeter gestisce anche il multi-threading e le multi-utenze, cioè può simulare richieste in parallelo, potenzialmente anche contemporanee, ed una concorrenza anche a livello di utenti.

Per questo studio ho effettuato dei test della durata di tre minuti ciascuno, durante i quali ho analizzato il comportamento dei sistemi effettuando ad ognuno di essi una media di 5 richieste al secondo nell'arco della durata del test. Si tenga presente che questo è un valore medio, in quanto le richieste non vengono effettuate in maniera uniforme. Ciò di cui si è certi è che al termine dei tre minuti saranno state inviate 900 richieste.

Procedendo allo stesso modo ho effettuato altri test incrementando gradualmente il numero di richieste al secondo, prima con 10, poi con 15, 20, 25 e 30.

Per non dilungarmi eccessivamente, illustrerò i risultati dei due sistemi per i due casi limite e quello centrale, quindi 5, 15 e 30 richieste al secondo.

10.2 Prestazioni

Durante lo studio dei comportamenti delle architetture, mi sono concentrato principalmente su due caratteristiche, ossia latenza e numero di errori al secondo, entrambe misure correlate al numero di richieste al secondo (R/s).

10.2.1 Test con 5 richieste al secondo

Analizziamo in prima fase la latenza media. Questa misura è fortemente influenzata dal numero di richieste che vengono recapitate al sistema, per cui è strettamente correlata all'andamento della quantità di richieste effettuate. Nel *Grafico 10.1* sono mostrati gli andamenti dei tempi di risposta media in entrambi i casi.

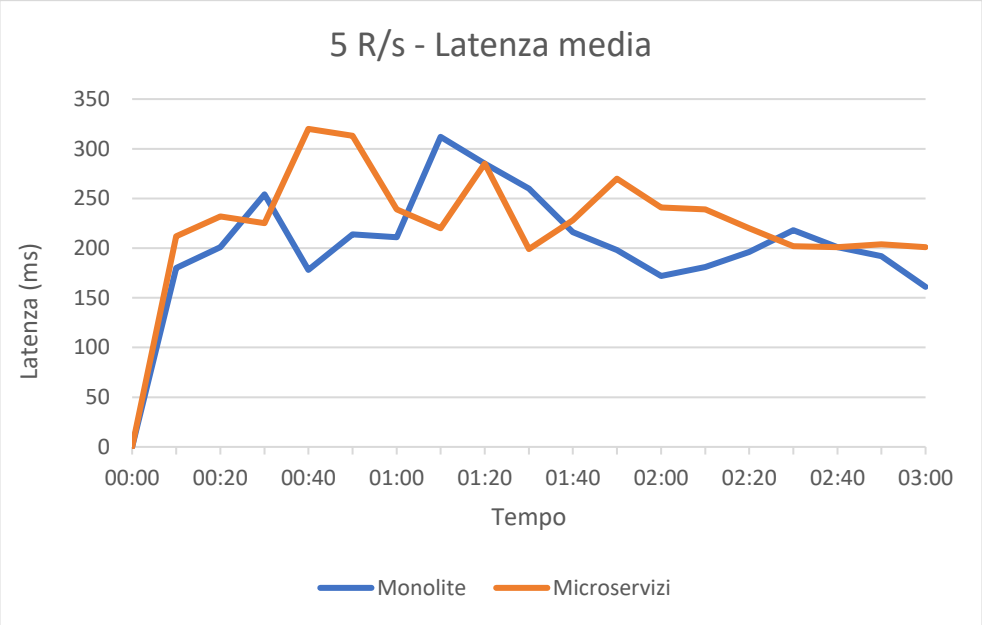

Grafico 10.1 – Rappresentazione dell'andamento della latenza media per architettura monolitica ed architettura a microservizi con 5 R/s (Elaborazione personale)

Ad un primo impatto visivo si evince subito che le due latenze medie sono assolutamente comparabili: le spezzate hanno quasi lo stesso picco di massimo ed entrambe hanno un andamento abbastanza altalenante, mentre il monolite ha in diversi casi dei picchi di minimo inferiori a quelli dei

microservizi. In questa fase quindi, con una bassa quantità di richieste, il primo riesce a rispondere leggermente più velocemente.

Nel caso del sistema monolitico, infatti, la latenza massima è stata di 312 ms, con una media durante l'intero test di 213 ms. Il sistema con architettura a microservizi invece ha registrato una latenza massima di 320 ms ed una latenza media di 236 ms.

Esaminiamo ora la quantità di errori verificatisi durante i tre minuti. Tale quantità sarà espressa come rapporto tra errori e secondi.

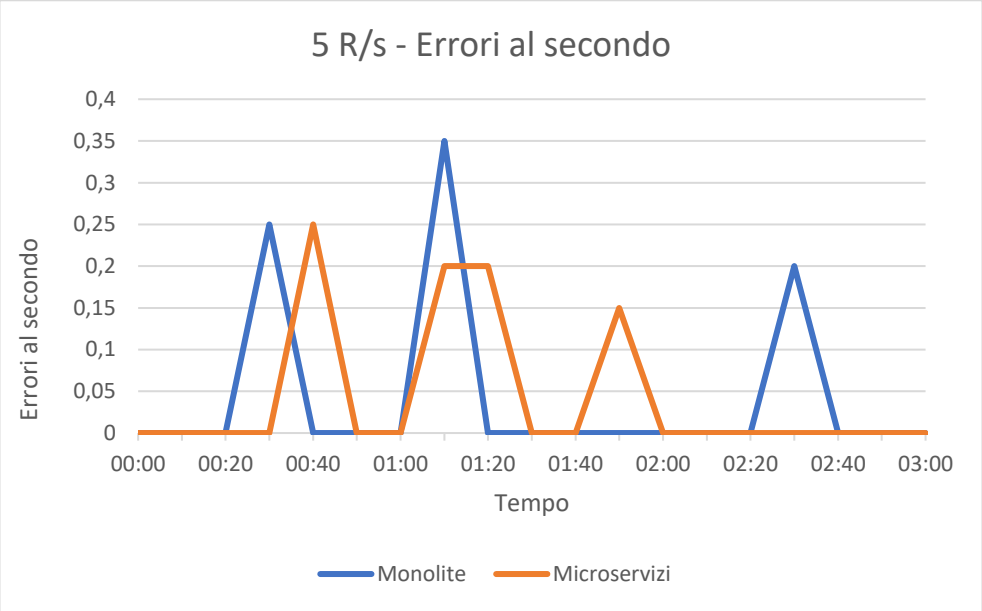

Grafico 10.2 – Rappresentazione dell'andamento degli errori al secondo per architettura monolitica ed architettura a microservizi con 5 R/s (Elaborazione personale)

Anche per questa misurazione gli andamenti nel *Grafico 10.2* sono molto simili. Su entrambi i sistemi si è verificato infatti lo stesso numeri di errori nell'arco del test, in questo caso pari a 8; la media è quindi di 0,044 errori al secondo.

Si tenga presente che tali errori sono stati tutti dovuti ad un "timeout", causati quindi dal fatto che i sistemi non siano riusciti a rispondere nel tempo massimo stabilito.

10.2.2 Test con 15 richieste al secondo

Passiamo al secondo test, durante il quale entrambi i sistemi ricevono nell'arco dei tre minuti 2700 richieste, con una media quindi pari a 15 richieste al secondo.

Analizziamo innanzitutto, come nel primo caso, la latenza media. Il *Grafico 10.3* rappresenta le due misurazioni ed in questo caso è subito evidente la differenza tra i due andamenti.

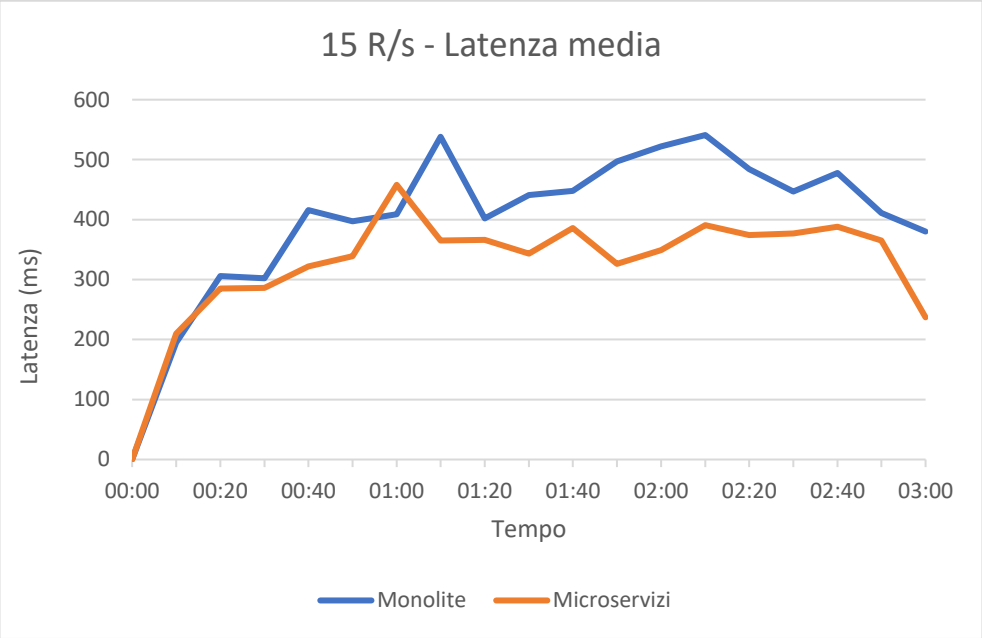

Grafico 10.3 – Rappresentazione dell'andamento della latenza media per architettura monolitica ed architettura a microservizi con 15 R/s (Elaborazione personale)

Notiamo come l'architettura a microservizi riesca a garantire dei tempi di risposta inferiori a quelli del monolite.

Quest'ultimo infatti ha risposto in media in un tempo pari a 423 ms, mentre la massima latenza registrata è stata pari a 541 ms. I microservizi si sono dimostrati molto più rapidi nelle risposte: la latenza massima è stata di 458 ms, mentre la media di 343 ms.

A questo punto analizziamo anche la quantità di errori con 15 R/s, per verificare se anche in questo caso sia evidente una differenza tra i due andamenti.

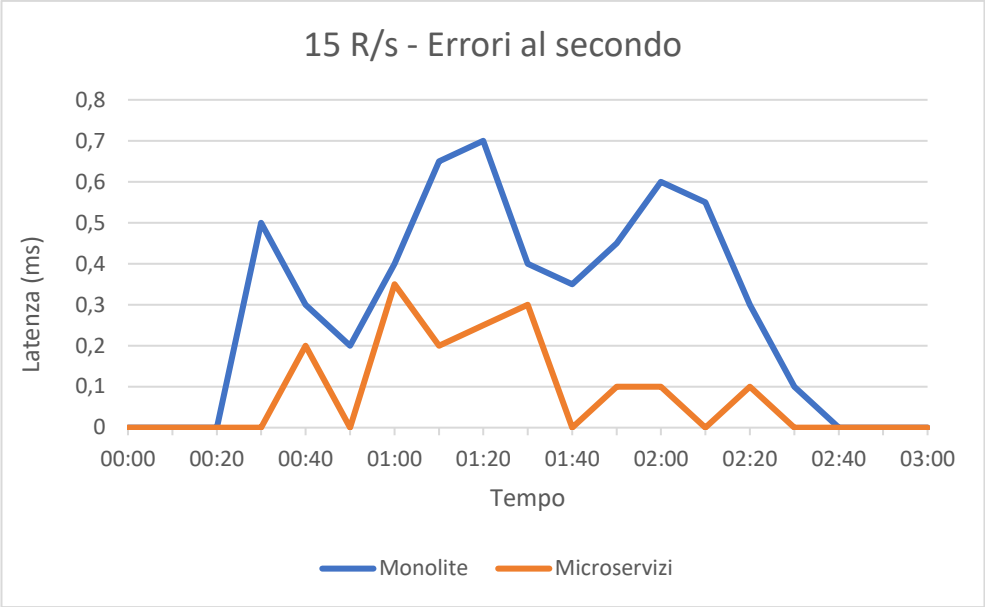

Grafico 10.4 – Rappresentazione dell'andamento degli errori al secondo per architettura monolitica ed architettura a microservizi con 15 R/s (Elaborazione personale)

Effettivamente dal *Grafico 10.4* la differenza è visibile anche per il numero di fallimenti dei sistemi. Il sistema monolitico ha generato 55 errori (media di 0,89 errori/secondo), contro i soli 16 del sistema a microservizi (0,30 errori/secondo).

10.2.3 Test con 30 richieste al secondo

Il test più "impegnativo" per i due sistemi si è completato effettuando 5400 richieste in tre minuti, sollecitando i sistemi per ben 30 R/s, con dei picchi vicini a 40 R/s.

Anticipo subito che questo è il test in cui si sono evidenziate le differenze maggiori, sia in termini di latenza media che di errori al secondo.

Come fatto in precedenza, prima di esporre i risultati, consultiamo i grafici degli andamenti.

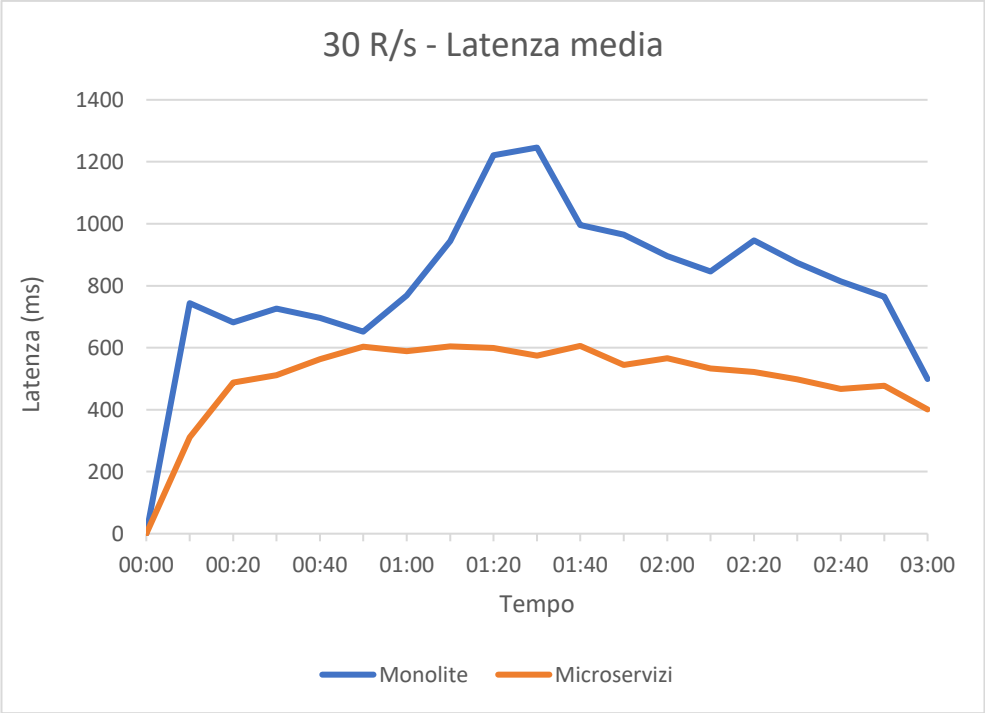

Grafico 10.5 – Rappresentazione dell'andamento della latenza media per architettura monolitica ed architettura a microservizi con 30 R/s (Elaborazione personale)

Il *Grafico 10.5* è molto esplicativo, e mostra una differenza importante tra i due sistemi. Il sistema monolitico in questo test è risultato essere molto meno efficiente, con una latenza media di risposta pari a 849 ms ed una massima di addirittura 1246 ms. La latenza dell'applicazione a microservizi è anch'essa aumentata, ma in maniera molto meno significativa: si è passati infatti ad una latenza media di 523 ms e di una massima di 606 ms. In questa fase quindi l'applicazione a microservizi risponde in circa metà del tempo impiegato dal monolite.

A questo proposito si noti anche la differenza della crescita della latenza media nei due casi. Nel passaggio tra 15 e 30 R/s, il tempo di risposta del monolite è praticamente raddoppiato. Lo stesso non può dirsi per i microservizi, i quali hanno anch'essi aumentato il ritardo nella risposta, ma in maniera decisamente minore (da 343 ms a 523 ms). Quindi al crescere della complessità, o comunque del numero di richieste, l'applicazione a microservizi perde efficienza in maniera abbastanza lineare, a differenza di quella monolitica, più vicina ad una curva esponenziale.

Anche il *Grafico 10.6* mostra importanti differenze, in questo caso sul numero di errori al secondo.

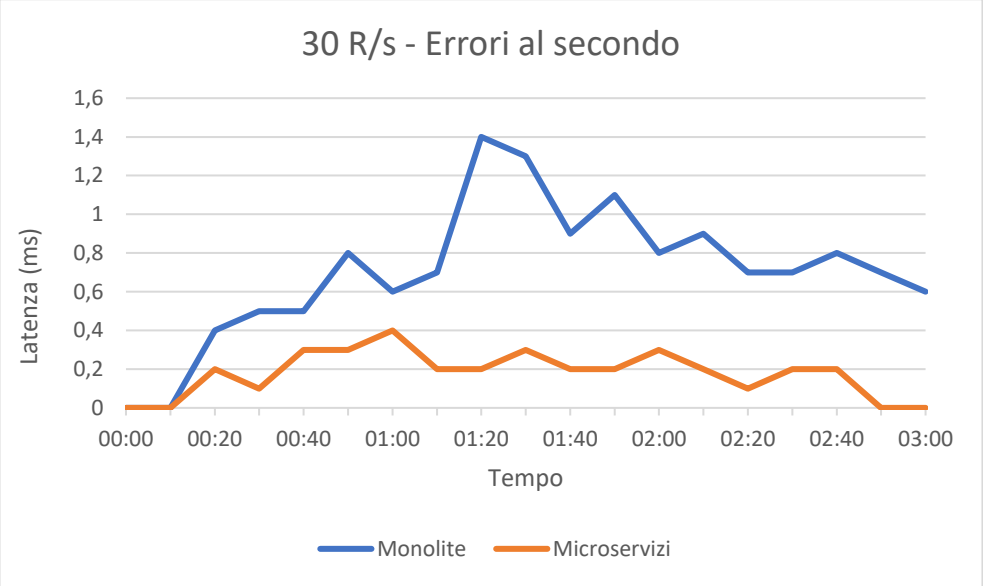

Grafico 10.6 – Rappresentazione dell'andamento degli errori al secondo per architettura monolitica ed architettura a microservizi con 30 R/s (Elaborazione personale)

Il sistema monolitico ha registrato addirittura una media di a 0,74 errori al secondo, pari a 134 errori nei tre minuti. Questo indica un fallimento di circa il 4,96% delle richieste effettuate nel test, una percentuale che inizia a diventare poco accettabile.

I numeri dell'architettura a microservizi sono invece molto più in linea con i precedenti: il sistema ha in questo caso registrato 34 errori, pari all'1,26% circa delle richieste.

10.3 Risorse

I paragrafi precedenti hanno mostrato come le prestazioni del monolite calino drasticamente all'aumentare delle richieste da servire. Il motivo principale della differenza di efficienza tra i due sistemi ad alti numeri di richieste è la differente scalabilità. Nel caso del monolite infatti questa non viene in alcun modo applicata, mentre nel caso dell'architettura a microservizi è stata settata una funzionalità di scalamento automatico.

All’avvio dei test, all’interno della PaaS sarà istanziato un POD per ciascun servizio. Per ognuno, prima della sua saturazione, la PaaS provvederà ad istanziare un nuovo POD a seconda delle esigenze, per poi deallocarlo quando non più necessario.

Nel caso del monolite invece, sappiamo che esso non può scalare in maniera automatica. L’unica possibilità sarebbe istanziarlo nuovamente, replicando il programma in esecuzione.

Per quanto riguarda il consumo di risorse, durante i vari test ho osservato che l’uso di CPU tra le due architetture è assolutamente paragonabile, motivo per cui nella trattazione tralascerò questo parametro. L’utilizzo di memoria invece può essere molto diverso, per questo motivo vale la pena affrontare il discorso più in profondità, confrontando il consumo di RAM delle applicazioni. Valuteremo quindi l’occupazione media di memoria nell’arco della durata di ogni test per ottenere dei sistemi che siano affidabili ed efficienti in egual misura.

Durante il primo test (5 richieste al secondo) i tre microservizi hanno utilizzato nell’arco dei tre minuti in media 318 MB, 356 MB e 330 MB, per un totale di 1004 MB di memoria. Ognuno di essi aveva istanziato un solo POD su ogni nodo, in tutti e tre i casi sufficiente a soddisfare le richieste.

Il sistema monolitico, invece, nella durata del test ha occupato in media 975 MB di RAM. Come visto in precedenza, nel primo caso con 5 R/s le prestazioni dei sistemi si sono rilevate molto simili, ed anche in questo caso il consumo di memoria è pressoché uguale dovendo entrambi allocare poco meno di 1 GB di RAM.

Dopo il secondo test, invece, ho iniziato a constatare le prime differenze. Il sistema monolitico inizia a non essere affidabile come l’architettura a microservizi, impostata per scalare automaticamente.

Il *Grafico 10.7* rappresenta in che modo sono scalati i microservizi, mostrando per ognuno il numero di POD istanziati durante il secondo test.

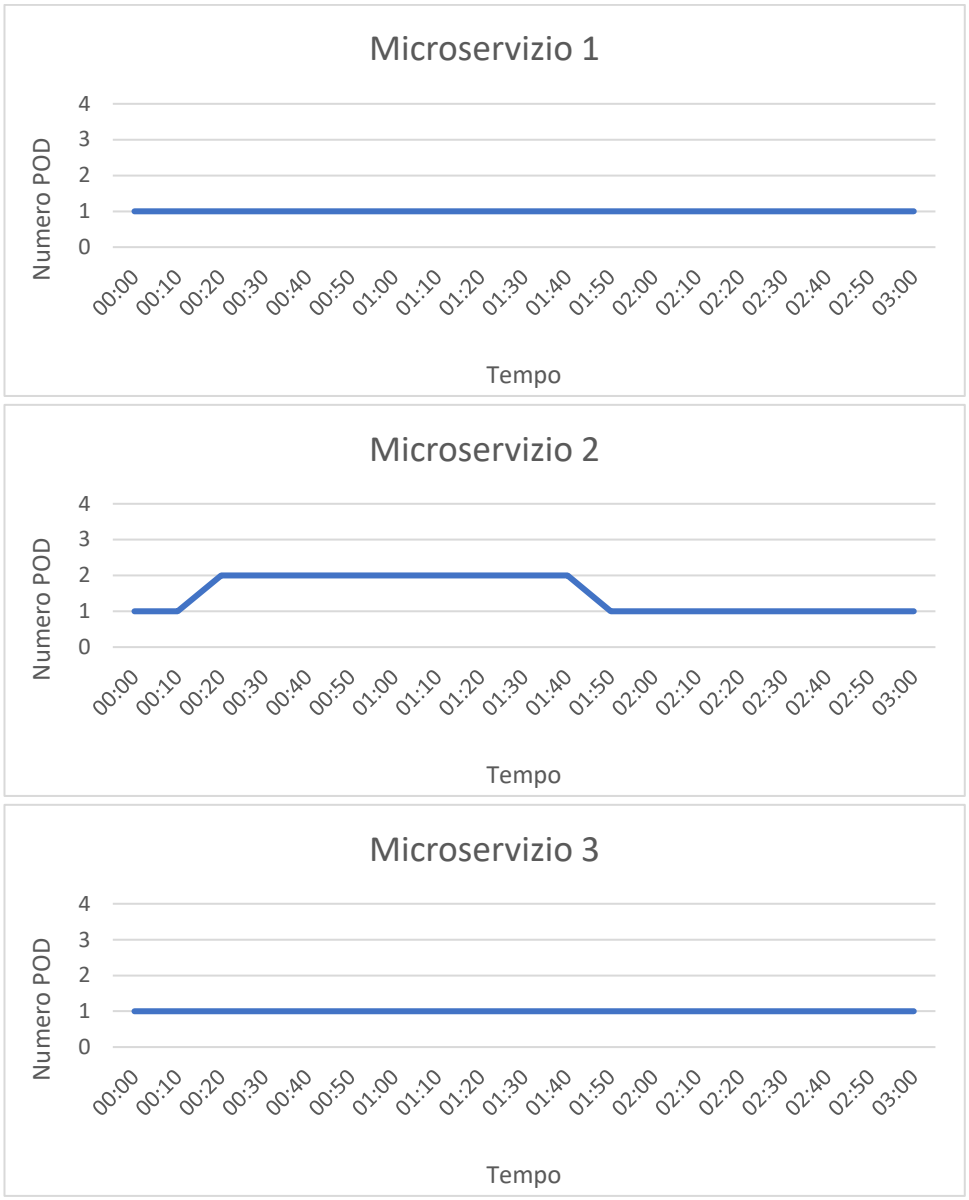

Grafico 10.7 – Rappresentazione del numero di POD allocati dai tre microservizi con 15 R/s
(Elaborazione personale)

Si noti subito come il secondo microservizio, prima di trovarsi in uno stato di "sofferenza", istanzi quasi fin da subito un secondo POD per servire più richieste. Quando per gestirle è nuovamente sufficiente solo uno, esso viene deallocato. Il motivo per il quale lo scalamento avvenga solo per il secondo servizio è che esso è molto più dispendioso, effettuando più operazioni in confronto alla sola interazione con un database, come nel caso del primo e del terzo.

Durante il secondo test quindi il consumo di memoria dei microservizi 1 e 3 è stato praticamente uguale a quello del primo test, e si è verificata un'occupazione aggiuntiva solo quando è stato istanziato il secondo POD. Ne consegue che l'allocazione media nell'arco dei tre minuti del secondo è aumentata di pochissimo, da 1004 MB a 1173 MB.

Durante Il terzo test osserviamo un comportamento simile rispetto al precedente. Anche in questo caso il Microservizio 1 ed il Microservizio 3 sono stati in grado di gestire le richieste allocando un solo POD.

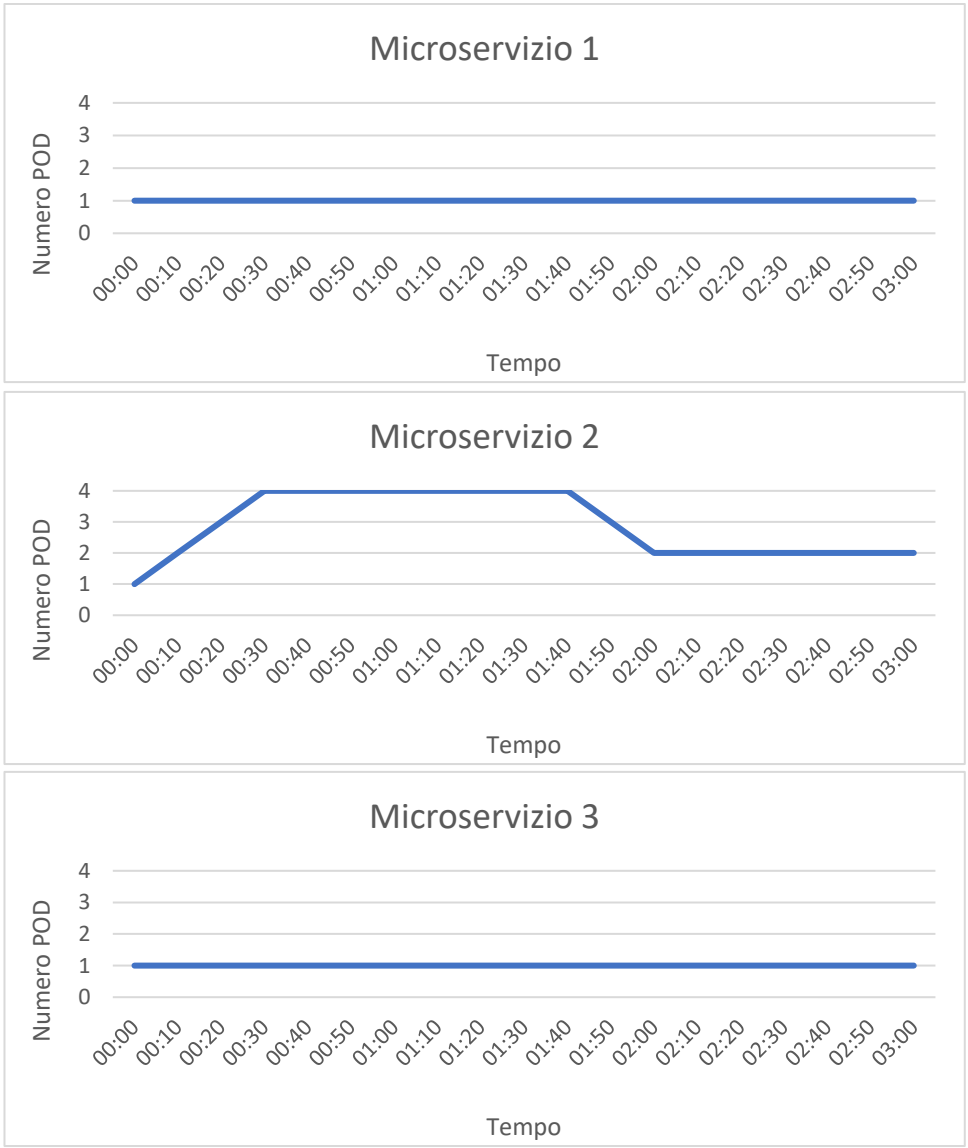

Grafico 10.8 – Rappresentazione del numero di POD allocati dai tre microservizi con 30 R/s
(Elaborazione personale)

Come si evince dal *Grafico 10.8* la PaaS ha allocato addirittura fino a quattro POD sul nodo del Microservizio 2, mettendolo nelle condizioni di soddisfare un numero di richieste ancora maggiore. La memoria utilizzata ovviamente aumenta, fino ad allocare in media circa 1678 MB.

Considerando i tre test, nel caso dell'applicazione a microservizi si è riusciti ad ottenere un buon livello di affidabilità ed efficienza. Per ottenere lo stesso nel monolite, nel test con 15 R/s avremmo dovuto quindi applicare il concetto di scalabilità orizzontale, andando a replicare un'istanza dell'applicazione. Nel terzo test, per soddisfare invece 30 R/s, sarebbe stato opportuno creare quattro istanze dell'applicazione. Il consumo di memoria quindi sarebbe raddoppiato nel secondo test e quadruplicato nel terzo.

Questo andamento è illustrato nel *Grafico 10.9*, il quale evidenzia proprio un accenno delle curve di crescita della quantità di risorse necessarie (in termini di memoria) ad ottenere un sistema affidabile al crescere dell'applicazione.

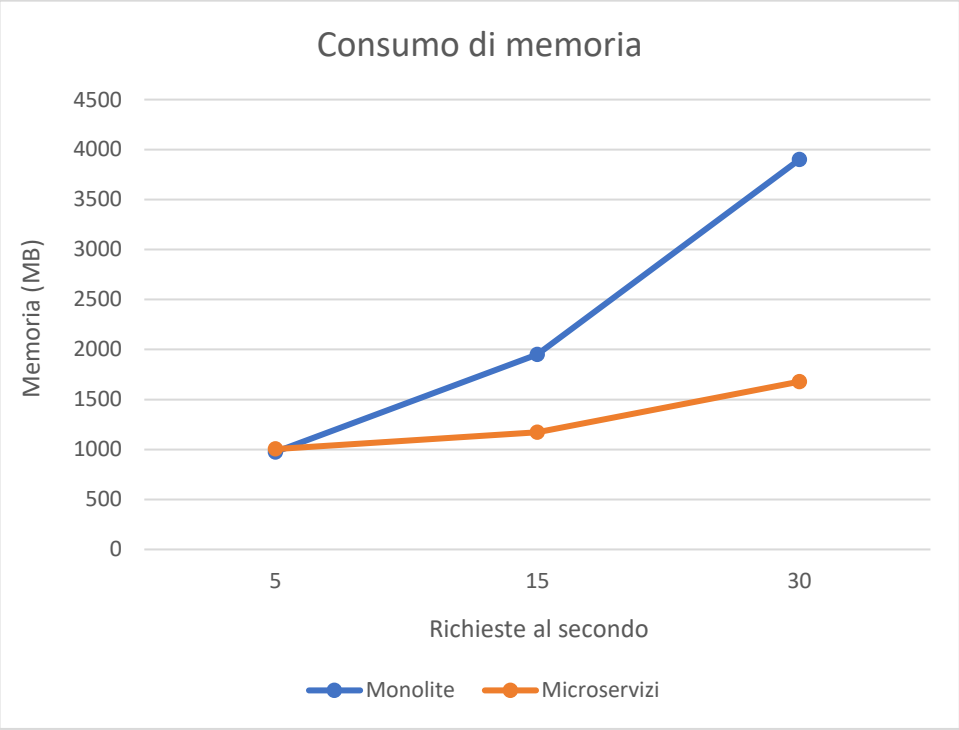

Grafico 10.9 – Rappresentazione dell'andamento della quantità di memoria utilizzata al crescere della complessità
(Elaborazione personale)

11 CONCLUSIONI

In base agli studi effettuati, alla vasta documentazione consultata ma soprattutto ai dati ricavati è possibile trarre le seguenti considerazioni.

L'aspetto più evidente è come né l'architettura a microservizi né i monoliti rispecchino la soluzione migliore in tutti i casi. Ad esempio, come evidenziato dal primo test, con 5 R/s l'architettura monolitica si è dimostrata più efficace. Tuttavia dopo il secondo o ancor di più dopo il terzo, all'aumentare della complessità, essa risultava sempre meno efficiente, contrariamente alla soluzione a microservizi, la quale invece garantiva migliore produttività all'aumentare della quantità di richieste.

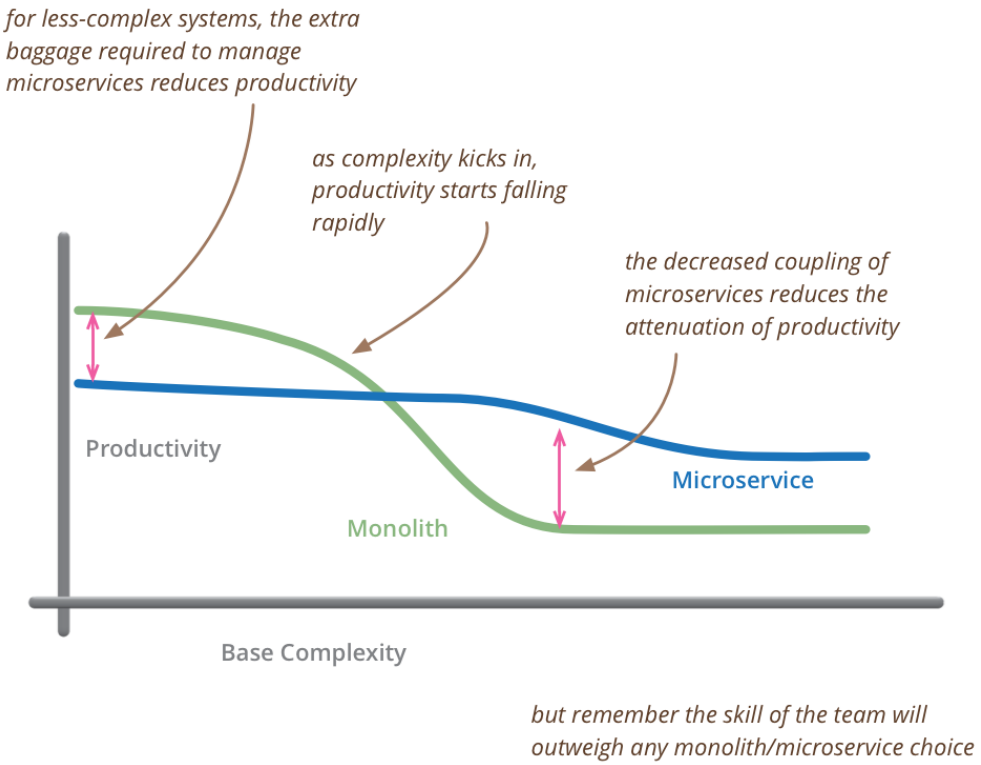

Figura 11.1 – Rappresentazione della produttività delle due architetture al variare della complessità [41]

La *Figura 11.1* riassume al meglio la più importante conclusione dello studio. Per sistemi poco complessi, che non devono soddisfare molte richieste e poco inclini al cambiamento, le architetture monolitiche possono rappresentare ancora una buona soluzione. Sicuramente in questo caso se si è in possesso di una buona applicazione monolitica non avrebbe senso investire risorse per trasformarla in un'applicazione a microservizi, in quanto non si avrebbero dei vantaggi tangibili.

In presenza di applicazioni molto grandi, che si dovranno interfacciare con un grande numero di utenti, come nel mio caso specifico, ha molto più senso investire su un'architettura a microservizi. Essa sul lungo termine garantirà sicuramente una maggiore efficienza e dei costi (in termini di risorse) minori.

12 BIBLIOGRAFIA

[1] DevOps is a culture, not a role!
Medium.com
Disponibile su: https://medium.com/@neonrocket/devops-is-a-culture-not-a-role-be1bed149b0 (Ultimo accesso: 10/03/2019)

[2] Aws Amazon – What is DevOps
Amazon.com
Disponibile su: https://aws.amazon.com/it/devops/what-is-devops/
(Ultimo accesso 02/11/2018)

[3] Aws Amazon – What is DevOps
Amazon.com
Disponibile su: https://aws.amazon.com/it/devops/what-is-devops/
(Ultimo accesso 02/11/2018)

[4] Guida alla Continuous Integration
Mokabyte
Disponibile su: http://www.mokabyte.it/2016/03/guidaci-1/ (Ultimo accesso: 10/03/2019)

[5] Introduction to Monolithic Architecture and MicroServices Architecture
Medium.com
Disponibile su: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63
(Ultimo accesso: 10/03/2019)

[6] Microservices Architecture: Il Pattern Architetturale Emergente Per Le Grandi Applicazioni Moderne
Lo Sviluppatore
Disponibile su: http://losviluppatore.it/microservices-architecture-il-pattern-architetturale-emergente-per-le-grandi-applicazioni-moderne/
(Ultimo accesso: 10/03/2019)

[7] Microservices
Martin Fowler
Disponibile su: https://martinfowler.com/articles/microservices.html
(Ultimo accesso: 10/03/2019)

[8] Introduction to Monolithic Architecture and MicroServices Architecture
Medium.com
Disponibile su: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63
(Ultimo accesso: 10/03/2019)

[9] Stile di architettura di microservizi
Microsoft
Disponibile su: https://docs.microsoft.com/it-it/azure/architecture/guide/architecture-styles/microservices (Ultimo accesso 10/03/2019)

[10] Cos'è l'infrastruttura cloud?
Hewlett Packard Enterprise
Disponibile su: https://www.hpe.com/it/it/what-is/cloud-infrastructure.html (Ultimo accesso 10/03/2019)

[11] Worldwide Spending on Cloud IT Infrastructure
IDC
Disponibile su:
https://www.idc.com/getdoc.jsp?containerId=prUS44358318 (Ultimo accesso: 10/03/2019)

[12] Gartner Forecasts Worldwide Public Cloud Revenue to Grow 21.4 Percent in 2018
Gartner
Disponibile su: https://www.gartner.com/en/newsroom/press-releases/2018-04-12-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-21-percent-in-2018 (Ultimo accesso: 10/03/2019)

[13] What is infrastructure as a service (IAAS)?
Redcentric
https://www.redcentricplc.com/resources/articles/what-is-iaas/ (Ultimo accesso: 10/03/2019)

[14] SaaS vs PaaS vs IaaS: What's The Difference and How To Choose
BMC
Disponibile su: https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/ (Ultimo accesso: 10/03/2019)

[15] Confronto tra schema API Gateway e comunicazione diretta da client a microservizio
Microsoft
Disponibile su: https://docs.microsoft.com/it-it/dotnet/standard/microservices-architecture/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern (Ultimo accesso: 10/03/2019)

[16] The Twelve-Factor App - Introduzione
The Twelve-Factor App
Disponibile su: https://12factor.net/it/ (Ultimo accesso: 10/03/2019)

[17] About
GIT
Disponibile su: https://git-scm.com/about (Ultimo accesso: 10/03/2019)

[18] Codebase
The Twelve-Factor App
Disponibile su: https://12factor.net/it/codebase (Ultimo accesso: 10/03/2019)

[19] Backing Service
The Twelve-Factor App
Disponibile su: https://12factor.net/it/backing-services (Ultimo accesso: 10/03/2019)

[20] Build, release, esecuzione
The Twelve-Factor App
Disponibile su: https://12factor.net/it/build-release-run (Ultimo accesso: 10/03/2019)

[21] Processi
The Twelve-Factor App
Disponibile su: https://12factor.net/it/processes (Ultimo accesso: 10/03/2019)

[22] Concorrenza
The Twelve-Factor App
Disponibile su: https://12factor.net/it/concurrency (Ultimo accesso: 10/03/2019)

[23] Circuit Breaker
Spring.io
Disponibile su: https://spring.io/guides/gs/circuit-breaker/ (Ultimo accesso: 10/03/2019)

[24] Circuit Breaker Architecture using Polly
Cloud & Mobile Blog
Disponibile su: https://cloudandmobileblogcom.wordpress.com/2017/04/15/circuit-breaker-architecture-using-polly/ (Ultimo accesso: 10/03/2019)

[25] 5 Blue-Green Deployment Best Practices for a Smooth Release
BlazeMeter
Disponibile su: https://www.blazemeter.com/blog/five-blue-green-deployment-best-practices-for-a-smooth-release (Ultimo accesso: 10/03/2019)

[26] CanaryRelease
Martinfowler.com
Disponibile su: https://martinfowler.com/bliki/CanaryRelease.html (Ultimo accesso: 10/03/2019)

[27] CanaryRelease
Martinfowler.com
Disponibile su: https://martinfowler.com/bliki/CanaryRelease.html (Ultimo accesso: 10/03/2019)

[28] CanaryRelease
Martinfowler.com
Disponibile su: https://martinfowler.com/bliki/CanaryRelease.html (Ultimo accesso: 10/03/2019)

[29] Vantaggi del Platform as a Service (PaaS) per lo sviluppo applicativo
Kiratech
Disponibile su: https://www.kiratech.it/blog/vantaggi-del-platform-as-a-service-paas-per-lo-sviluppo-applicativo (Ultimo accesso: 10/03/2019)

[30] Spring Framework Overview
Spring
Disponibile su: https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html (Ultimo accesso: 11/03/2019)

[31] What Is Spring Boot?
DZone
Disponibile su: https://dzone.com/articles/what-is-spring-boot (Ultimo accesso: 11/03/2019)

[32] What Is Spring Boot?
DZone
Disponibile su: https://dzone.com/articles/what-is-spring-boot (Ultimo accesso: 11/03/2019)

[33] Developer Survey Results 2017
StackOverflow
Disponibile su: https://insights.stackoverflow.com/survey/2017#technology-programming-languages (Ultimo accesso: 11/03/2019)

[34] Google App Engine
Google
Disponibile su: https://cloud.google.com/appengine/ (Ultimo accesso: 11/03/2019)

[35] Spring Data JPA
Spring
Disponibile su: https://spring.io/projects/spring-data-jpa (Ultimo accesso: 11/03/2019)

[36] I vantaggi dei container Linux
RedHat
Disponibile su: https://www.redhat.com/it/topics/containers (Ultimo accesso: 12/03/2019)

[37] Cos'è un container Linux?
RedHat
Disponibile su: https://www.redhat.com/it/topics/containers/whats-a-linux-container (Ultimo accesso: 12/03/2019)

[38] Cos'è Docker?
RedHat
Disponibile su: https://www.redhat.com/it/topics/containers/what-is-docker (Ultimo accesso: 12/03/2019)

[39] Cos'è Docker?
RedHat
Disponibile su: https://www.redhat.com/it/topics/containers/what-is-docker (Ultimo accesso: 12/03/2019)

[40] Apache JMeter
Apache
Disponibile su: https://jmeter.apache.org/ (Ultimo accesso: 12/03/2019)

[41] Microservices Resource Guide
Martinfowler.com
Disponibile su: https://www.martinfowler.com/microservices/ (Ultimo accesso: 12/03/2019)