

# **ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA**

---

**SCUOLA DI INGEGNERIA E ARCHITETTURA**

Corso di Laurea in Ingegneria Informatica

Dipartimento di Informatica - Scienza e Ingegneria

**TESI DI LAUREA**

in

Reti di Calcolatori – T

## **Performance di Architetture EventMesh a supporto di Microservizi**

Tesi di Laurea di:

Leonardo Dominici

Relatore:

Chiar.mo Prof. Ing. Antonio Corradi

Correlatori:

Dott. Andrea Sabbioni  
Dott. Alessandro Calvio

---

Anno Accademico 2022-2023

---

Sessione I

# Sommario

<b>Introduzione .....</b>	<b>4</b>
<b>1 Cloud Computing.....</b>	<b>5</b>
1.1 Definizione .....	5
1.2 Storia.....	6
1.3 Vantaggi.....	6
1.4 Svantaggi .....	7
1.5 Modelli di cloud deployment.....	9
1.5.1 Public cloud .....	9
1.5.2 Private cloud .....	9
1.5.3 Hybrid cloud .....	10
1.5.4 Community cloud .....	10
1.5.5 Multi Cloud.....	11
1.6 Modelli di servizi cloud .....	11
1.6.1 IaaS .....	12
1.6.2 PaaS .....	12
1.6.4 Saas.....	13
1.7 Virtualizzazione .....	14
1.7.1 Tipo 1 – Livello host.....	14
1.7.2 Tipo 2 – Livello SO .....	14
<b>2 Microservizi .....</b>	<b>16</b>
2.1 Architettura del singolo microservizio .....	17
2.2 Vantaggi.....	17
2.3 Svantaggi .....	19
2.4 Containerizzazione .....	20
2.5 Orchestrazione .....	21
2.6 Docker.....	22
2.6.1 Componenti.....	23
2.6.2 Docker Swarm “Classic” .....	26
2.7 Docker Swarm Mode.....	27
<b>3 Kubernetes.....</b>	<b>29</b>
3.1 Componenti.....	30
3.1.1 Control Plane .....	31

3.1.2 Nodi Worker .....	34
3.1.3 Addon .....	35
3.2 Objects .....	36
3.2.1 Pod .....	37
3.2.2 Deployment.....	40
3.2.3 Service .....	44
3.3 Network .....	48
3.4 Storage .....	52
<b>4 ServiceMesh e EventMesh.....</b>	<b>55</b>
4.1 Definizione ServiceMesh.....	56
4.2 Componenti ServiceMesh.....	59
4.2.1 Sidecar .....	59
4.2.2 Control Plane .....	61
4.2.3 Data Plane.....	61
4.3 Implementazioni ServiceMesh .....	62
4.4 Istio ServiceMesh .....	63
4.4.1 Data Plane.....	64
4.4.2 Control plane .....	64
4.4.3 Modelli di deployment.....	67
4.5 EventMesh .....	72
4.6 Componenti EventMesh .....	74
4.6.1 Sidecar Proxy o EventMesh Nodes .....	75
4.6.2 Event Broker.....	76
4.6.3 Strumenti di log .....	77
4.7 Implementazioni EventMesh .....	78
4.8 Apache EventMesh.....	79
4.8.1 EventMesh Runtime .....	79
4.8.2 EventMesh Store.....	81
4.8.4 EventMesh Schema Registry.....	82
4.8.5 EventMesh Metrics.....	83
<b>5 Progetto e implementazione di un Event Mesh .....</b>	<b>84</b>
5.1 Cluster Kubernetes.....	85
5.2 Architettura .....	86
5.2.1 Producer.....	87
5.2.2 Consumer .....	89
5.2.3 Proxy.....	89

5.2.4 EventBroker.....	93
5.3 Interazione componenti .....	95
5.4 Benchmark.....	96
5.4.1 Send Singola .....	97
5.4.2 Send Parallelia .....	100
5.5 Deployment Apache EventMesh .....	103
<b>Conclusioni .....</b>	<b>104</b>
<b>Bibliografia .....</b>	<b>107</b>

# Introduzione

Negli ultimi 15 anni il mercato del cloud computing è nato, si è sviluppato molto rapidamente e ha visto una forte impennata in termini di utilizzatori e fatturato complessivo. Sfruttare componenti o infrastrutture realizzate e mantenute dai cloud provider risulta particolarmente comodo per organizzazioni con esigenze specifiche in termini economici, di scalabilità e di affidabilità.

Un pattern architetturale il cui sviluppo è stato influenzato dallo sviluppo del cloud computing è il pattern a microservizi. In contrapposizione a un'applicazione monolitica, un'applicazione a microservizi è composta da una serie di piccole unità logiche indipendenti, responsabili di singole funzionalità applicative, che comunicano fra loro per offrire e sfruttare servizi.

In alcuni casi gli applicativi a microservizi possono essere composti da svariate migliaia di microservizi. In scenari del genere è difficile garantire una comunicazione ben coordinata ed efficiente, e spesso diventa necessario inserire logica volta alla comunicazione fra microservizi in componenti che dovrebbero solo realizzare la logica applicativa.

Strumenti come ServiceMesh ed EventMesh, permettono di costruire layer infrastrutturali facilmente controllabili che permettono un'interazione efficiente fra i microservizi senza intaccare le unità che realizzano la logica di business.

In architetture ad eventi, EventMesh risulta essere una soluzione particolarmente interessante per gestire in modo dinamico le interazioni fra microservizi.

EventMesh sfrutta i proxy, piccoli componenti che fungono da gateway per le comunicazioni uscenti ed entranti dai microservizi. I proxy in collaborazione con gli event broker si occupano di veicolare la comunicazione fra microservizi attraverso l'uso di eventi, garantendo una comunicazione asincrona, con basse latenze e permettendo ai microservizi di essere estremamente disaccoppiati.

Durante questa tesi è stata realizzata un'implementazione di EventMesh, sfruttando Kubernetes e inserendo i proxy, sviluppati in NodeJS, nello stesso pod dei microservizi rendendoli così sidecar proxy. Usando Kafka come event broker sono poi stati effettuati test per verificare le performance dell'implementazione analizzando l'entità della latenza introdotta da ogni componente e stressando il sistema attraverso Apache Bench.

# 1 Cloud Computing

## 1.1 Definizione

Il cloud computing è una metodologia di accesso a risorse computazionali estremamente popolare e adottato dalla grande maggioranza delle aziende operanti non solo in settori tecnologici

Secondo IBM il cloud computing può essere definito come:

*Cloud computing is on-demand access, via the internet, to computing resources—applications, servers (physical servers and virtual servers), data storage, development tools, networking capabilities, and more — hosted at a remote data center managed by a cloud services provider (or CSP). [ ... ] (IBM, n.d.)*

Gli attori in gioco all'interno di un sistema basato su cloud computing sono molteplici. Internet innanzitutto è il tramite utilizzato per trasferire informazioni e fornire servizi ai fruitori del cloud computing, non a caso lo sviluppo di internet in termini di velocità, affidabilità e costo ha fortemente impattato l'impiego di servizi di cloud computing.

Questo approccio alle risorse informatiche sposta e distribuisce la capacità di calcolo, di storage e di banda su molteplici macchine spesso non gestite o direttamente controllate dal fornitore del servizio. Le organizzazioni che offrono un servizio tramite cloud computing si affidano ad un attore esterno al fine di delegare alcune parti del proprio servizio, queste parti delegate vengono solitamente gestite attraverso reti di macchine in cooperazione fra loro.

Sono 3 le principali caratteristiche di un sistema di cloud computing (Varghese, 2019):

- Permettere l'utilizzo di capacità computazionali come fossero un bene di consumo, vendute per tempo di utilizzo o quantità delle risorse usate.
- La condivisione di risorse utilizzate da più persone nello stesso contempo, realizzato grazie alla virtualizzazione.
- Accesso alle risorse attraverso internet.

## 1.2 Storia

Il cloud computing può considerarsi una evoluzione dell'utility computing, come venne descritto già nel 1961 dal Prof. John McCarthy, durante il suo discorso per il centenario dell'MIT di Boston dicendo:

*Computing may someday be organized as a **public utility** just as the telephone system is a public utility. [ ... ]. Each subscriber needs to pay only for the capacity he actually uses, but he has access to all programming languages characteristic of a very large system [ ... ] Certain subscribers might offer service to other subscribers [ ... ] The computer utility could become the bases of a new and important industry. (Garfinkel, 2011)*

AWS (Amazon Web Services, la sussidiaria di Amazon che si occupa di fornire servizi di cloud computing) nel 2006 lanciò EC2 (Elastic Compute Cloud) permettendo ai propri utenti di utilizzare macchine virtuali create e gestite da AWS. Nel 2008 Google lanciò una beta per 10.000 sviluppatori presentando Google App Engine, un prodotto per ospitare tutto lo stack di applicazioni web su server Google, questa fu una significativa differenza rispetto ad Amazon che all'epoca offriva principalmente 3 servizi molto modulari e non per forza usati in cooperazione (EC2, S3 per storage, SimpleDB come database) (Arrington, 2008). Nel 2010 Microsoft seguì l'esempio di Google e AWS con la sua divisione di cloud computing Microsoft Azure, i 4 principali strumenti offerti da Azure erano Windows Azure che permetteva agli sviluppatori di eseguire web app ASP.NET, Azure Blob come servizio di storage, SQL Azure come servizio di database (MSV, 2020) e Azure Service Bus, ovvero un message broker con code di messaggi e semantica pub-sub.

## 1.3 Vantaggi

Indipendentemente dalle dimensioni e dal tipo di azienda i vantaggi del cloud computing sono molteplici:

- Investimento iniziale ridotto:

Il cloud computing non richiede un investimento iniziale paragonabile a quello di una architettura on-premise, piccole aziende con poco capitale possono iniziare ad offrire il proprio servizio ai clienti senza sostanziali investimenti iniziali.

- Facilità d'impiego e riduzione della manutenzione:

Non possedendo e non gestendo in prima persona l'hardware utilizzato non è necessario avere un'infrastruttura in grado di fornire in modo affidabile energia e connettività ai propri server. Molti servizi di cloud computing offrono inoltre soluzioni dedicate ad utilizzi specifici, invece che una macchina intera si può decidere di acquistare solo le istanze di applicativi (ad esempio Google Big Query all'interno di Google Cloud Platform), non sarà necessario occuparsi di aggiornamenti del sistema, sicurezza a livello di rete e setup non direttamente legati all'istanza.

- Scalabilità:

Delegando la gestione e il provisioning delle macchine utilizzate ad un provider con ampie capacità in termini di risorse computazionali, sarà facile scalare i servizi aumentando le capacità computazionali disponibili sulle macchine utilizzate. Questo aspetto è molto importante soprattutto per piccole organizzazioni senza le capacità tecniche e finanziarie per scalare rapidamente on-premise.

- Pay-per-use:

La possibilità di pagare solo per le risorse effettivamente utilizzate è uno degli aspetti chiave del cloud computing, permettendo di ridurre significativamente i costi fissi.

In sintesi, i vantaggi sono garantiti dalla grande disponibilità di capacità computazionali offerte dai cloud provider. La conoscenza e l'esperienza necessaria per gestire servizi di questo tipo è inoltre difficile da sviluppare in organizzazioni il cui prodotto principale è un altro.

## 1.4 Svantaggi

Come qualsiasi approccio e tecnologia anche il cloud computing porta con sé un numero di svantaggi che in alcuni casi d'uso possono essere significativi. Ecco i principali:

- Dipendenza dalla connessione internet:

Internet è strettamente legato all'impiego del cloud computing in quanto rappresenta il mezzo attraverso il quale i server comunicano con l'utilizzatore finale. Assenza di internet o basse performance della connessione impediscono un efficiente utilizzo di questi servizi.

- Sicurezza:

Non ospitando i server in prima persona e non conoscendo nel dettaglio tutte le procedure di sicurezza applicate dai cloud provider, è possibile essere esposti a vulnerabilità senza esserne coscienti. Il controllo totale delle macchine è sempre in mano al provider del servizio, quantomeno dal punto di vista dell' hardware e della connettività.

- Costo:

Sebbene i costi siano estremamente variabili, il cloud computing abbatte i costi fissi, aumentando però i costi variabili rispetto a soluzioni on-premise. Un'accurata comparazione viene fatta da Cameron Fisher (Fisher, 2018) in cui evidenzia come il costo per l'infrastruttura in scenari on-premise abbia un picco nella fase iniziale dell'adozione, mentre il cloud computing sposti la maggioranza delle spese più avanti nel tempo, rendendo però queste spese costanti e complessivamente maggiori.

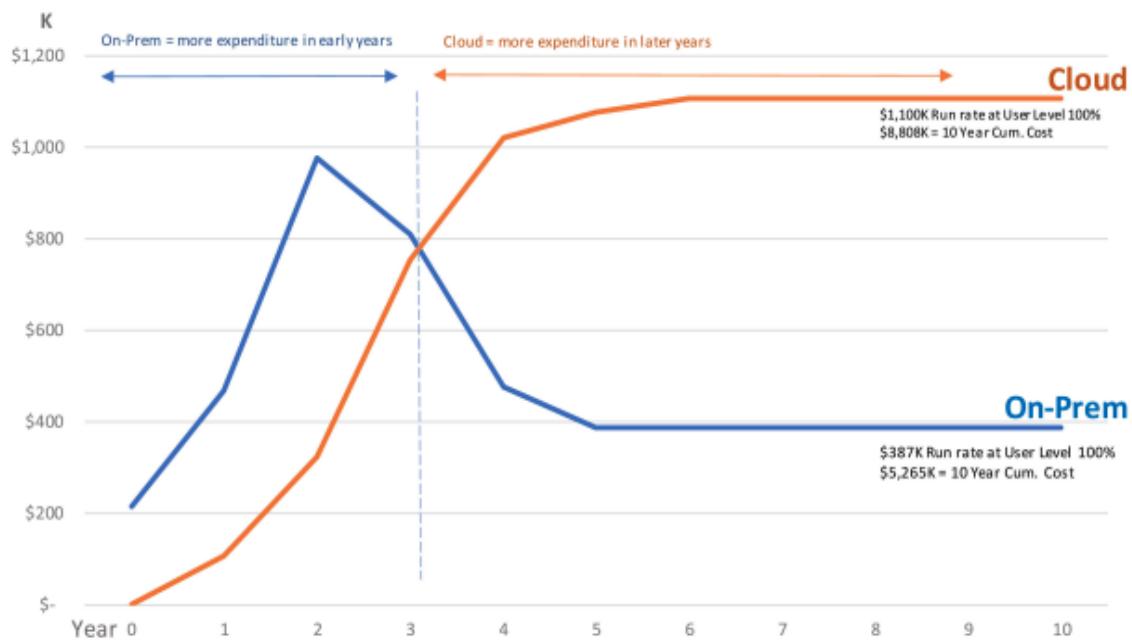


Immagine 1 - Cloud contro On-Premise comparazione costi decennale (Fisher, 2018)

- Vendor lock-in:

Una volta adottato un servizio di cloud computing può essere difficile abbandonarlo in breve tempo. Nel caso di AWS, ovvero il fornitore di servizi cloud più grande al mondo con circa il 34% del market share (Synergy Research Group, 2023), le migrazioni verso altri provider sono rese particolarmente complicate. Questo impedisce spesso ai clienti di spostarsi verso servizi migliori o più economici, nel caso in cui AWS decidesse di aumentare i prezzi o ridurre i servizi offerti.

La maggior parte degli svantaggi sono causati dalla dipendenza che si ha nei confronti del provider di servizi.

Affidandosi a terzi per la gestione dei propri servizi si cede parte della propria libertà di scelta, fidandosi e auspicandosi che il provider faccia le scelte migliori per il cliente.

## **1.5 Modelli di cloud deployment**

I modelli di cloud deployment specificano il tipo di ambiente cloud, discriminando in base a proprietario, dimensione, accesso e scopo del cloud.

### **1.5.1 Public cloud**

Il cloud è disponibile al pubblico e chiunque può comprare e utilizzare i servizi offerti dal cloud provider. La maggior parte dei servizi vengono pagati tramite formule pay-per-usage (Google Cloud, n.d.), a questa categoria appartengono i più conosciuti servizi di cloud computing come quelli offerti da AWS, Microsoft Azure e GCP (Google Cloud Platform).

I servizi di public cloud offrono risorse contenute in pool di risorse distribuite in data center spesso dislocati in tutto il mondo, l'accesso avviene sempre tramite internet. Questi servizi sono spesso molto grandi e organizzati, offrono servizi molto prestanti con pochi limiti in termini di risorse massime acquistabili.

### **1.5.2 Private cloud**

I private cloud si differenziano dai public cloud per il fatto di non essere aperti al pubblico ma riservati solo a un gruppo limitato e definito di utenti. I servizi di questo tipo di cloud vengono offerti via internet o attraverso una intranet dell'organizzazione che detiene il datacenter (Microsoft

Azure, n.d.). Il datacenter è on-premise, ospitato quindi dall'azienda che offre ai propri dipendenti il servizio.

Tagliando fuori il provider terzo questa configurazione permette di avere standard di sicurezza più stringenti e controllabili, oltre ad avere costi solo legati all'effettivo utilizzo di risorse e non legati alla necessità di profitto del provider. Spesso i private cloud vengono scelti in caso si debba lavorare con dati finanziari, medici o militari dal momento che le regolamentazioni in merito a dati sensibili sono molto stringenti. Questo servizio richiede però di effettuare un grande investimento iniziale e di disporre di persone e infrastrutture capaci di gestire e mantenere uno o più datacenter. Spesso il private cloud viene anche chiamato Internal Cloud o Corporate Cloud (IBM, n.d.).

### **1.5.3 Hybrid cloud**

L'hybrid cloud prevede l'utilizzo di un mix di differenti public cloud, private cloud e sistemi on-premise.

Si tratta di una configurazione molto diffusa poiché permette di sfruttare gli aspetti positivi di differenti tipi di cloud.

L'hybrid cloud è spesso uno step intermedio obbligato dal momento che una qualsiasi transizione da soluzioni on-premise a soluzioni in cloud implica uno spostamento graduale del carico di lavoro (Google Cloud, n.d.). È inoltre molto comune disporre di un'infrastruttura on-premise per gestire il traffico regolare mentre si sposta il carico di lavoro in cloud durante i periodi in cui ci sono picchi di richieste (cloud burst), così da sfruttare sempre al massimo la propria infrastruttura senza incappare in limitazioni.

Per impiegare l'hybrid cloud è necessario considerare l'overhead volto a gestire e coordinare le operazioni su cloud differenti, che spesso non potrebbero essere automaticamente compatibili.

### **1.5.4 Community cloud**

Il community cloud ha vita quando un gruppo di organizzazioni appartenenti ad uno stesso gruppo (ad esempio molti ministeri della stessa nazione) condividono un servizio di cloud computing. L'infrastruttura può essere creata e mantenuta dal gruppo stesso o appoggiandosi a provider di terze parti come un normale public cloud provider.

Se il servizio è ospitato on premise senza affidarsi a provider terzi, gli standard di sicurezza sono più controllabili e spesso più alti rispetto a quelli di un public cloud.

Unendo differenti organizzazioni si avranno orari di picco differenti, necessità differenti e una capacità di assorbire aumenti di richieste maggiore rispetto ad un private cloud di dimensioni più ridotte.

Il costo è maggiore di quello di un public cloud e spesso può essere difficile coordinare tante organizzazioni nell'adottare determinate tecnologie o nell'aderire alle medesime regole.

### 1.5.5 Multi Cloud

Si tratta di un approccio che implica l'utilizzo di 2 o più public o private cloud. Sebbene sia spesso considerato sinonimo di hybrid cloud non si tratta dello stesso concetto. Un hybrid cloud è caratterizzato da un'interconnessione di public e private cloud al fine di svolgere lo stesso compito. Nel multicloud invece si usano i servizi di più provider per compiere diversi tipi di operazioni. Un hybrid cloud può anche essere considerato multi cloud se include risorse di un private cloud e da almeno 2 public cloud differenti (Google Cloud, n.d.).

Spesso vengono impiegate architetture multi cloud per evitare di essere totalmente dipendenti da un solo provider e riuscire a sfruttare i servizi più vantaggiosi di ogni provider.

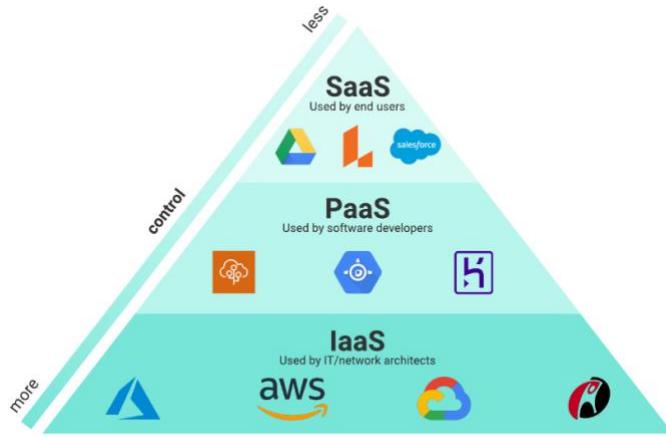
## 1.6 Modelli di servizi cloud

I servizi di cloud computing possono essere erogati in modi differenti lasciando o meno all'utente la libertà di decidere come comportarsi e cosa fare a diversi livelli dello stack applicativo. Ogni modello di cloud service prevede differenti gradi di libertà, controllo e gestione.

Spese in public cloud per utenti finali in milioni di dollari:

	2021	2022	Crescita YoY
PaaS	89.910	110.677	+23%
SaaS	146.326	167.107	+14%
IaaS	90.894	115.740	+27%
Altri (BPaaS, DaaS)	57.011	62.666	+10%

Tabella 1 - Spese in servizi public cloud per utenti finali (Gartner, 2022)



*Immagine 2 - Modelli di servizio cloud e livello controllo sul servizio*

I 3 modelli di servizio più comunemente usati sono Infrastructure as a Service (IaaS), Platform as a Service (PaaS) e Software as a Service (SaaS).

### 1.6.1 IaaS

È il livello che garantisce la gestione più a basso livello.

Il provider fornisce una Virtual Machine (VM) vuota al cliente, offrendo però connettività, corrente elettrica, storage, protezione da DoS, e in generale tutta l'infrastruttura IT necessaria a mantenere la VM accesa e online. Il cliente è responsabile del setup della macchina inclusa la configurazione del sistema operativo e tutti gli applicativi da esso usati.

### 1.6.2 PaaS

È un modello di distribuzione di servizi cloud che fornisce un ambiente di sviluppo e di esecuzione completo per le applicazioni. In un'offerta PaaS, il provider si occupa, oltre agli aspetti già offerti nella IaaS, anche di fornire servizi e strumenti che semplificano il ciclo di vita delle applicazioni. Questi servizi includono l'allocazione automatica delle risorse, il bilanciamento del carico, il monitoraggio delle prestazioni, la scalabilità automatica, la gestione delle patch e degli aggiornamenti del software. Il cliente si occupa di sviluppare e configurare le proprie applicazioni utilizzando le risorse e gli strumenti forniti dal provider. Il cliente può concentrarsi sulla scrittura del codice, la configurazione dell'ambiente di esecuzione e la gestione dei dati.

Risulta particolarmente comodo per sviluppatori nel facilitare lo sviluppo, il deployment e il mantenimento di applicazioni. Evita di gestire il sistema operativo, SDK, runtime e strumenti di sviluppo (Google Cloud, n.d.).

I vantaggi principali sono la diminuzione del tempo per rilasciare e pubblicare software, lasciando il cliente totalmente libero di decidere cosa e come sviluppare il proprio servizio.

#### 1.6.4 SaaS

SaaS è la forma più completa di servizi di cloud computing, consiste nella fornitura di un'intera applicazione gestita da un provider. L'utente interagisce con l'applicazione offerta spesso tramite API o dashboard (Gmail o Outlook offrono a tutti gli effetti un SaaS).

Riduce al minimo le capacità e il personale necessario per offrire un servizio e mantenerlo funzionante.

In questi servizi è spesso ridotta la libertà di personalizzazione, la condivisione dello stesso ambiente di esecuzione può causare problemi di sicurezza e spesso porta le aziende a trovarsi in una situazione di vendor lock-in.

On-Premises	IaaS	PaaS	FaaS	SaaS
Applications	Applications	Applications	Functions	Applications
Data	Data	Data	Data	Data
Runtime	Runtime	Runtime	Runtime	Runtime
Middleware	Middleware	Middleware	Middleware	Middleware
O/S	O/S	O/S	O/S	O/S
Virtualization	Virtualization	Virtualization	Virtualization	Virtualization
Servers	Servers	Servers	Servers	Servers
Storage	Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking	Networking

You Manage
Other Manages

Immagine 3 - Responsabilità dello stack in vari modelli di servizi cloud

## 1.7 Virtualizzazione

La virtualizzazione è fondamentale per il cloud computing ed è alla base della condivisione delle risorse dei sistemi informatici.

*La virtualizzazione è un processo che permette un più efficiente utilizzo dell'hardware ed è la base del cloud computing (IBM, n.d.).*

La virtualizzazione permette di emulare il comportamento di una macchina reale attraverso un software di virtualizzazione, crea una rappresentazione logica delle risorse della macchina, senza essere direttamente legata allo strato fisico ma essendo mediata nella connessione dal virtualizzatore.

Le componenti chiave della virtualizzazione sono 3: hardware, hypervisor, macchine virtuali.

L'hardware si interfaccia con l'hypervisor senza entrare direttamente in contatto con le macchine virtuali.

Gli hypervisor forniscono una visione logica delle risorse fisiche di una macchina reale e si occupano di mediare le relazioni fra le macchine virtuali e l'hardware sottostante astraendone però l'implementazione.

Ci sono due tipi di hypervisor, o virtualizzatori, quelli di tipo 1 e di tipo 2, anche rispettivamente chiamati virtualizzatori a livello host e a livello sistema operativo.

### 1.7.1 Tipo 1 – Livello host

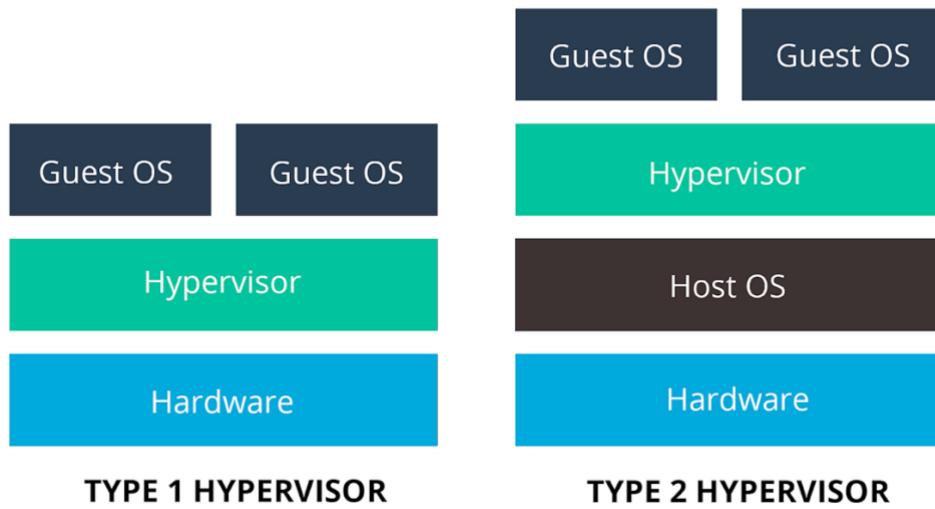
L'hypervisor funge da sistema operativo e viene installato come tale subito al di sopra dell'hardware, questo approccio permette maggiori prestazioni, maggiore sicurezza e controllo di tutti i parametri di vita dell'hypervisor stesso e delle macchine virtuali, viene preferito nella maggior parte delle situazioni soprattutto in scenari di produzioni in cui le performance e l'affidabilità giocano un ruolo chiave.

### 1.7.2 Tipo 2 – Livello SO

L'hypervisor è un programma installato sopra al sistema operativo preesistente, hypervisor di questo tipo sono eseguiti come programmi dal sistema operativo.

Spesso sono usati negli scenari in cui serve disporre di due o più sistemi operativi allo stesso momento sulla medesima macchina per esigenze di sviluppo o di retrocompatibilità di applicativi (IBM, n.d.).

Questo approccio aggiunge passaggi intermedi e conseguente latenza dal momento che, ogni risorsa va acquisita passando attraverso il sistema operativo della macchina host e non passando solo attraverso il virtualizzatore come nel caso degli hypervisor di tipo 1.



*Immagine 4 - Suddivisione dello stack in hypervisor di tipo 1 (livello host) e tipo 2 (livello SO)*

## 2 Microservizi

L’architettura a microservizi consiste in uno stile architettonico per lo sviluppo di applicazioni. I microservizi permettono a una grande applicazione di essere suddivisa in molteplici piccole parti indipendenti fra loro, ognuna delle quali si occupa di un compito specifico di cui è responsabile. Per fornire risposta alle richieste dei clienti un’applicazione a microservizi può contattare i molteplici microservizi di cui è composta al fine di generare una risposta da inviare all’utente (Google Cloud, n.d.).

Questa architettura si contrappone alle architetture monolitiche, da poco meno di 10 anni l’architettura a microservizi è divenuta estremamente popolare e rappresenta uno degli standard architettonici più impiegati. L’utilizzo di questo pattern architettonico permette di sfruttare appieno le capacità del cloud computing, per questo motivo l’architettura a microservizi è considerata un’architettura cloud-native (IBM, n.d.).

I microservizi sono spesso divisi in base alle funzionalità che offrono, team differenti sono responsabili di microservizi differenti ed ogni servizio risponde al principio della “single responsibility”.

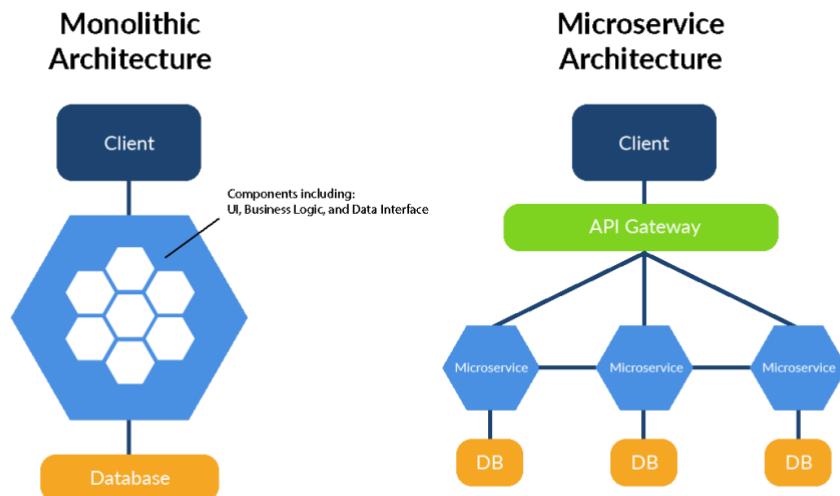


Immagine 5 - Organizzazione architettura a microservizi comparata ad una monolitica

## **2.1 Architettura del singolo microservizio**

Un'applicazione monolitica non può essere arbitrariamente frammentata in microservizi senza alcun tipo di accorgimento. I microservizi devono disporre di componenti necessarie a renderli indipendenti e debolmente interconnessi, le componenti necessarie in ogni microservizio sono:

- Logica di business
- API
- Gestore delle dipendenze
- Database interno in caso il microservizio sia stateful e non si appoggi a microservizi esterni da cui ottenere lo stato
- Servizio di monitoraggio

Nel caso di monoliti le interazioni avvengono fra componenti racchiuse in un unico ambiente in cui il mediatore è il runtime o il sistema operativo. La comunicazione fra microservizi comporta invece un overhead dal momento che qualsiasi comunicazione deve essere inoltrata attraverso la rete, passare attraverso le API, ed essere ricevuta e interpretata dal destinatario. Molti di questi compiti sono delegati all'infrastruttura di gestione dei microservizi, mentre altri sono realizzati all'interno dei microservizi.

## **2.2 Vantaggi**

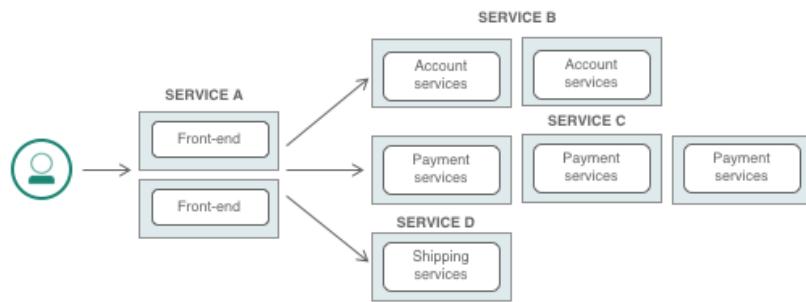
I vantaggi dei microservizi derivano soprattutto dalla forte indipendenza fra le varie componenti, questo permette di lavorare con unità logiche più contenute, che svolgono meno compiti e hanno meno responsabilità se comparate ad un unico monolite.

Alcuni dei principali vantaggi dei microservizi sono:

- Scalabilità:  
Molti servizi, soprattutto se direttamente utilizzati da utenti non rispondono solo a carichi costanti nel tempo, bensì, sono soggetti a picchi di carico in corrispondenza di fenomeni esterni al servizio e spesso non prevedibili. Esempi di questi fenomeni sono i picchi di traffico nei sistemi di messaggistica durante eventi sportivi o nei picchi d'utilizzo di servizi di streaming durante gli orari non lavorativi. Per un servizio è importante garantire continuità di servizio e stabilità delle performance, anche quando le richieste di un sistema cambiano rapidamente e in modo drastico. Il metodo principale per assecondare aumenti

(e successivi cali) di traffico è scalare il servizio. Con scalare un servizio si intende l'atto di aumentare prestazioni e capacità complessiva del servizio per far fronte ad aumenti di carico, o al contrario, di ridurre il numero di istanze per risparmiare capacità computazionali e quello che ne consegue.

Solo alcuni microservizi utilizzati all'interno di una applicazione potrebbero aver bisogno di essere scalati, senza il bisogno di scalare tutti gli altri. Tramite load balancer è poi facile bilanciare il carico in ingresso destinato alle istanze di uno stesso microservizio in modo da distribuire le richieste per garantire performance accettabili a tutti gli utenti.



*Immagine 6 - Differenti numeri di repliche per ogni microservizio interfacciato con il frontend [19]*

- Indipendenza di sviluppo e deployment:

Lavorando su un microservizio alla volta, operando modifiche solo sulla sua ristretta porzione di codice è più facile apportare modifiche in modo efficace. La possibilità di effettuare rilasci graduali permette poi di sostituire gradualmente le istanze di un microservizio al fine di accorgersi subito di eventuali errori. In caso di errori è possibile fermare il rollout, bloccando il deployment delle nuove istanze di microservizio aggiornato.

- Affidabilità:

Errori anche critici all'interno di un microservizio non hanno impatti sugli altri microservizi. In caso un microservizio riscontri problemi, verranno limitate le funzionalità offerte da quel servizio, ma essendo i microservizi suddivisi per funzionalità di business, solo quella data funzionalità sarà impattata. La facilità di deployment e interconnessione dei microservizi permette inoltre di distribuirli su diversi server rendendo il servizio più resistente a problematiche legate al fallimento di componenti hardware.

- Flessibilità di sviluppo:

Ogni microservizio può essere realizzato in modo fondamentalmente diverso dai microservizi con cui collabora. Microservizi differenti possono essere realizzati da gruppi di lavoro diversi impiegando differenti tecnologie. Affinchè i microservizi collaborino efficientemente assieme è necessaria una chiara interfaccia a cui fare richieste tramite API o tramite il protocollo di interazione definito, questo meccanismo permette di astrarre da linguaggi o ambienti di runtime usati.

L'approccio a microservizi è fortemente legato all'impiego del cloud computing, servizi veramente distribuiti, ridondanti e affidabili sono difficilmente implementabili senza un approccio fortemente modulare come quello introdotto dai microservizi.

### **2.3 Svantaggi**

Gli svantaggi introdotti dai microservizi sono molteplici, ecco i principali:

- Complessità:

Il coordinamento e l'interazione necessaria alla comunicazione fra loro i microservizi può diventare complicata e difficoltosa da gestire, una buona documentazione e una conoscenza dell'infrastruttura in cui operano i microservizi è fondamentale

- Latenza delle comunicazioni:

Far interagire componenti su macchine diverse o addirittura in datacenter diversi può facilmente far la latenza e generare molto traffico all'interno della rete.

- Complessità dei test:

I test da effettuare nei sistemi a microservizi sono maggiori dal momento che è necessario testare le funzionalità end-to-end, i singoli microservizi e le interazioni fra i diversi microservizi.

- Integrità dei dati:

Avendo componenti distribuite può essere complesso mantenere uno stato condiviso da tutti i microservizi in tutti i momenti, questo problema può essere mitigato usando transazioni e tecniche di gestione e coordinamento dello stato condiviso.

Un approccio distribuito implica la presenza di problematiche comuni a tutti i sistemi distribuiti. Problematiche come il coordinamento e l'integrazione fra servizi possono essere mitigate da una documentazione puntuale e dall'impiego di infrastrutture ad hoc per la gestione di microservizi. La flessibilità garantita da questo approccio può essere controproducente per il sistema se non gestita con modi e metodologie opportune.

## 2.4 Containerizzazione

Con containerizzazione intendiamo il raggruppamento di codice e di tutti i relativi componenti necessari al suo funzionamento come librerie, framework e altre dipendenze, in modo che risultino isolate in un proprio contenitore che prende il nome di container (RedHat, 2021).

La containerizzazione consente di avere cellule di codice autosufficienti che necessitano solo di un ambiente di runtime comune (come Docker) per essere eseguite, qualsiasi altra dipendenza viene soddisfatta tramite risorse interne al container. Queste cellule di codice funzionante possono essere facilmente migrate fra infrastrutture diverse permettendo lo sfruttamento di un'infrastruttura distribuita come quella introdotta dal cloud computing.

A differenza delle VM, nei container il kernel è condiviso con la macchina su cui il container viene eseguito.

Se le VM hanno dimensioni nell'ordine dei gigabyte i container sono nell'ordine dei megabyte, dal momento che a differenza delle VM i container non devono contenere un OS ma solo l'applicativo e le sue dipendenze da lanciare nel suo ambiente di esecuzione (RedHat, 2021).

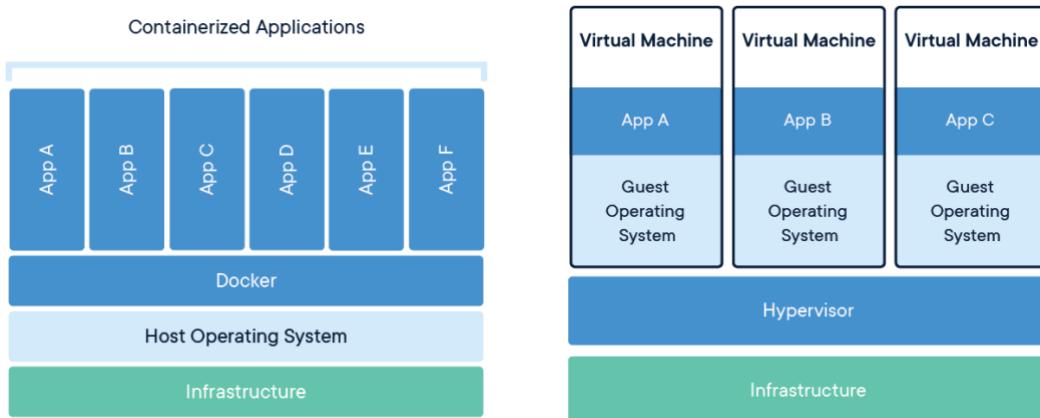


Immagine 7 - Comparazione fra containerizzazione e utilizzo VM (Docker, n.d.)

In conclusione, i microservizi sono programmi che soddisfanno e realizzano una funzionalità logica ben definita e si interfacciano con altri microservizi offrendo API come fossero un servizio del tutto indipendente. I container permettono di contenere tutte le dipendenze e tutti gli ambienti necessari per eseguire il programma. Date queste caratteristiche, i container sono ottimi strumenti per contenere i microservizi, permettendone lo spostamento e l'esecuzione senza necessità di avere un ambiente compatibile all'esterno, dal momento che tutte le dipendenze necessarie sono presenti all'interno del sistema.

## 2.5 Orchestrazione

Definiamo orchestrazione la configurazione, la gestione e il coordinamento automatici di sistemi informatici, applicazioni e servizi. È una metodologia che aiuta i gruppi IT a gestire più facilmente attività e flussi di lavoro complessi (RedHat, 2019).

L'orchestrazione è un tipo di automazione rivolta in particolar modo ai processi di rilascio e deployment di software, alla gestione delle risorse e alla gestione del carico.

Quando si parla di orchestrazione ci si riferisce solitamente ai processi e agli strumenti usati per la gestione dei container.

In un sistema molto flessibile e composto da molti componenti dipendenti fra loro come nel caso dei microservizi è fondamentale coordinare le operazioni nel migliore dei modi al fine di sfruttare tutte le opportunità offerte da questi strumenti.

Alcuni esempi di orchestrazione sono: scale up automatico dei servizi, rimpiazzo di microservizi falliti e gestione del rollout di aggiornamenti in modo sicuro (Docker, n.d.).

Un sistema di orchestrazione efficiente deve essere in grado di recepire istruzioni tramite un linguaggio formalizzato e di interfacciarsi con il container engine (nel caso dei container) al fine di operare sulle unità logiche in gioco. Al fine di coordinare centinaia o migliaia di container è necessario impiegare strumenti strutturati per evitare di gestirli manualmente, automatizzare questa gestione è fondamentale al fine di non effettuare errori.

Nella maggior parte dei casi si parla di orchestration di container, vediamo quindi quali sono i principali compiti svolti da un orchestratore di container:

- Configurazione e provisioning dei container:

Si usano file (YAML o JSON di solito) volti a definire le caratteristiche di un container e le relazioni che deve mantenere con il resto dell'ambiente.

- Deployment dei container:  
Soprattutto in sistemi composti da più macchine è fondamentale capire su quali nodi eseguire i container in base a quante e quali risorse sono disponibili su ogni nodo.
- Scaling dei container:  
In base al carico di lavoro su di un container, o gruppi di container, può essere necessario aumentare il numero di istanze o spostare alcune istanze su altri nodi.
- Service discovery:  
È importante avere un modo per ottenere i riferimenti di altri container per sfruttarne gli eventuali servizi offerti. Serve quindi fornire strumenti come i DNS, per referenziare i container disponibili.

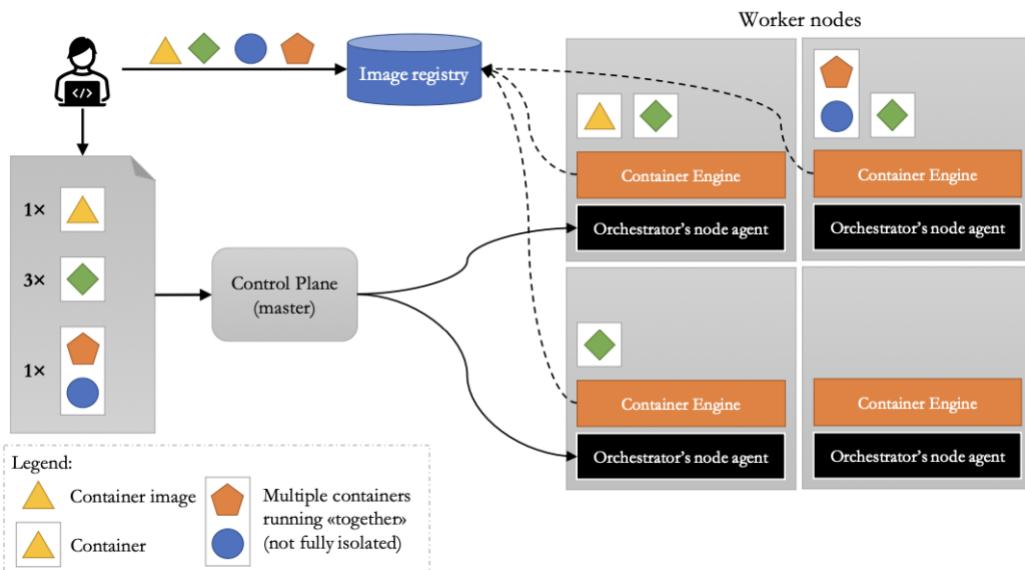


Immagine 8 - Overview di un sistema di orchestrazione generico nella gestione di una applicazione (Culturra, 2022)

## 2.6 Docker

Docker è una delle principali piattaforme per lavorare con i container. Offre due componenti necessari all'utilizzo di questa tecnologia, i Docker container e il Docker Engine ovvero il runtime al di sopra del quale possono essere usati i container. Docker è sviluppato dalla Docker Inc. che oltre a distribuire gratuitamente Docker e ad aver reso open source importanti parti di codice come containerd, si occupa di vendere versioni Professional di Docker che offrono funzionalità avanzate

rispetto alla versione Community come, ad esempio, tool di orchestrazione avanzata per gestire configurazioni multi-cloud.

*Docker è una piattaforma aperta per lo sviluppo, la condivisione e l'esecuzione di applicativi software, Docker permette di separare le applicazioni dall'infrastruttura in modo da sviluppare software rapidamente. Con Docker è possibile gestire l'infrastruttura come se fosse una applicazione. Sfruttando le metodologie di Docker per condividere, testare ed effettuare deployment velocemente è possibile ridurre significativamente il tempo che passa fra la scrittura del codice e la sua esecuzione in ambienti di produzione (Docker, n.d.).*

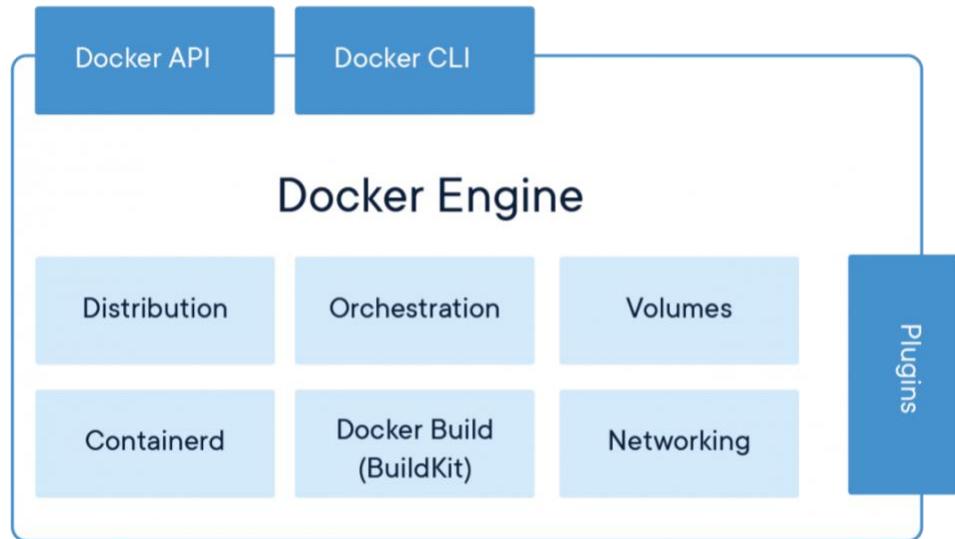
Docker permette di sviluppare e spostare applicazioni su container, rende il container Docker l'unità di distribuzione per il rilascio e il test, e permette di eseguire il container gestendone tutti gli aspetti.

Con oltre 18 milioni di sviluppatori che usano Docker e oltre 13 miliardi di download di Docker Images, Docker si dimostra leader nel settore della containerizzazione e prova la validità dell'approccio a container e la sua efficacia in molteplici settori.

## 2.6.1 Componenti

Le principali componenti di Docker sono:

- Docker Engine:  
È la componente fondamentale di Docker, è il runtime environment che crea ed esegue i container. Si tratta di un demone chiamato dockerd che può essere controllato tramite API o CLI. Il componente principale del Docker Engine è il container runtime containerd, che si occupa principalmente di gestire il ciclo di vita dei container.



*Immagine 9 - Componenti e interfacce del Docker Engine (Docker , n.d.)*

- **Docker Image:**

Sono file read-only eseguibili, facilmente portabili, contenenti tutte le direttive per creare i container e definire quale software eseguire al loro interno. Spesso le immagini sono costruite a partire da altre immagini. Ad esempio l'immagine di un web server può essere costruita a partire da un'immagine di Ubuntu all'interno del quale viene installato e configurato un web server. È quindi possibile sfruttare un'immagine già esistente a cui aggiungere solo le modifiche necessarie a realizzare l'immagine con le funzionalità desiderate.

- **Docker Container:**

È l'istanza eseguibile di un'immagine. I container seguono un ciclo di vita che viene controllato dalle Docker API.

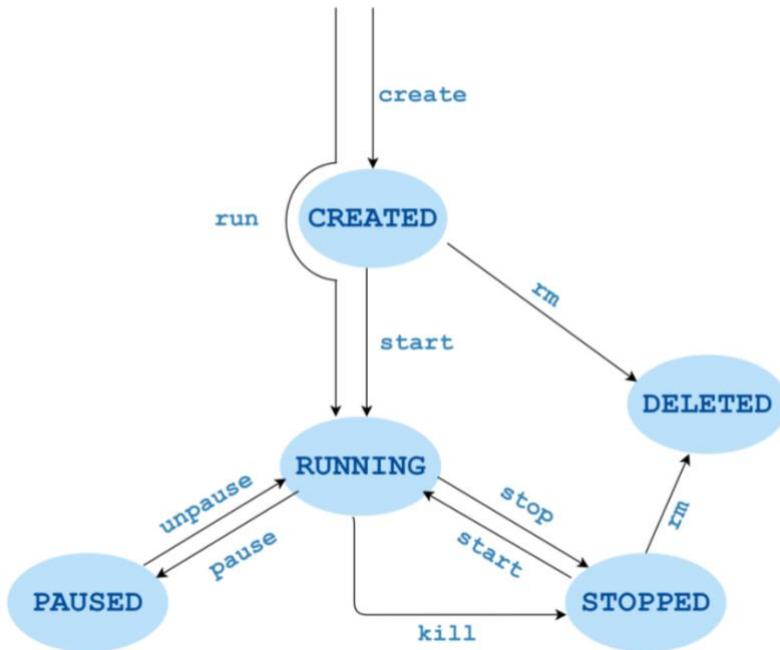


Immagine 10 - Ciclo di vita di un container Docker

- Docker Registry:

Il sistema usato da Docker per salvare, gestire e versionare le immagini dei container. Può essere pubblico come Docker Hub o privato e utilizzato da singoli o organizzazioni.

## Basic taxonomy in Docker

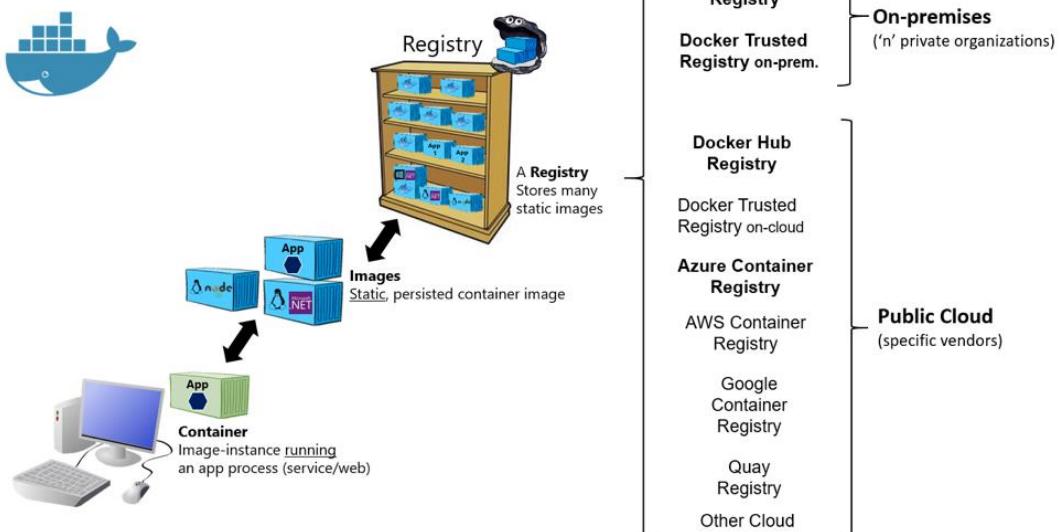


Immagine 11 - Tassonomia e impiego dei Docker Registry (Microsoft Learn, 2022)

- Dockerfile:  
Ogni container si basa su un Dockerfile che descrive l'immagine Docker da eseguire, le variabili d'ambiente, i riferimenti a componenti o servizi esterni, le posizioni di file o aspetti legati alla rete.
- Docker Compose:  
È un tool a linea di comando in grado di interpretare file YAML che definiscono applicazioni basate su una cooperazione di più container.
- Docker Networking:  
È il componente che permette la comunicazione fra container generando reti interne a Docker con diverse caratteristiche e specifiche definibili dall'utente.
- Docker Storage:  
È il modulo Docker che si occupa di fornire la persistenza ai container. Dal momento che i container sono entità effimere è necessario sfruttare componenti esterne ai container per il mantenimento dello stato.

### **2.6.2 Docker Swarm “Classic”**

Docker Swarm Classic rappresenta il primo sistema di clustering nativo per Docker, ha il compito di trasformare una pool di host Docker in un unico host virtuale. Pubblicato nel 2014, dal 2021 lo strumento non è più mantenuto a causa dei numerosi svantaggi presenti rispetto al resto dei prodotti sul mercato. Swarm Classic possiede uno Swarm Manager unico che si occupa di gestire tutto il cluster, schedulando i task sui nodi worker in base alle risorse disponibili e offrendo funzionalità di monitoraggio. Il controllo del cluster attraverso Swarm viene realizzato attraverso chiamate alle API o attraverso un’interfaccia web.

Oltre alle stringenti limitazioni in merito al numero di nodi, Swarm defica anche di alcune funzionalità fondamentali soprattutto in ambienti di produzione come il load balancing e i rolling updates oltre a richiedere un processo di installazione particolarmente complesso.

La risposta di Docker Inc. a tutto ciò è stata integrare del tutto Docker Swarm all’interno della versione 1.12 di Docker rilasciata nel 2016.

## 2.7 Docker Swarm Mode

Docker Swarm Mode ha sostituito Docker Swarm Classic a partire dal 2016. Il passaggio da Docker Swarm Classic alla Docker Swarm Mode ha portato notevoli vantaggi. Integrando il sistema di orchestrazione direttamente in Docker, e non tramite uno strumento esterno, è stato possibile ridurre sensibilmente la complessità di installazione e aggiungere diverse funzionalità non presenti nella versione Classic. Le principali funzionalità sono:

- Gestione del cluster integrato:  
Gestione del cluster completamente delegata a Docker Engine.
- Approccio dichiarativo:  
Per definire lo stato desiderato del sistema di container generato.
- Multi host networking:  
Permette di creare ed esporre i container su VLAN interne.
- Service discovery:  
Grazie all'integrazione di un DNS interno è possibile per un container conoscere le altre istanze presenti nel cluster.
- Load balancing:  
Realizzato esponendo le porte dei servizi all'esterno e definendo regole di load balancing interne.
- Autenticazione TLS:  
Ogni node dello Swarm può usare autenticazione e cifratura TLS la cui gestione è delegata a Docker Swarm.
- Rolling updates:  
Impiegati per aggiornare i container in modo graduale ed effettuare rollback in caso di problemi con i nuovi container.

Un Docker Swarm è composto da molteplici host Docker, alcuni degli host sono eseguiti in swarm modo e fungono da manager mentre gli host restanti sono eseguiti come swarm services e fungono da worker per ospitare i container in esecuzione (Docker, n.d.). Swarm può gestire servizi, che sono insieme di task (gruppi di container Docker attivi), impostazioni di rete, impostazioni di storage, numero di repliche e altri parametri. Data una configurazione fornita in ingresso Swarm

lavora al fine di rispettare le specifiche della configurazione. Ad esempio, nel caso di fallimento di un nodo, Swarm riavvia i container presenti su quel nodo distribuendoli su altri nodi attivi.

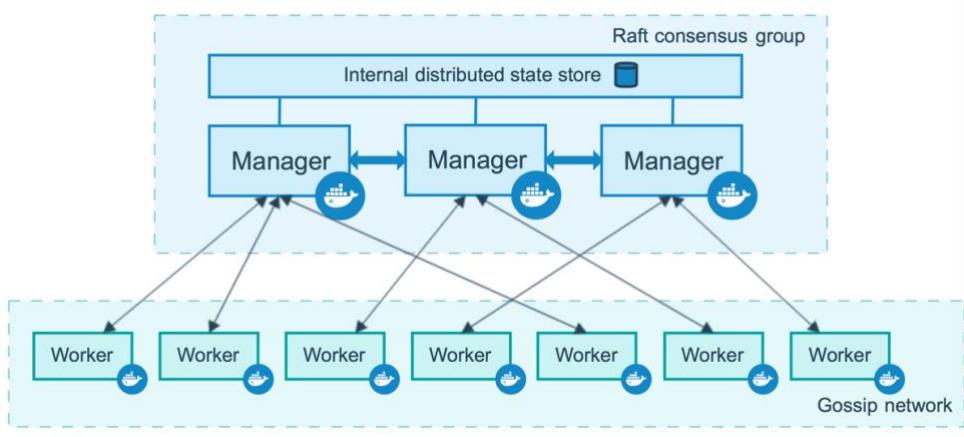


Immagine 12 - Architettura e nodi in Docker Swarm Mode [29]

Il principale ruolo dei manager è quello di schedulare e organizzare il deployment dei task sui nodi worker disponibili. I manager condividono uno stato interno, gestito tramite un algoritmo di consenso chiamato Raft. Raft elegge un leader che viene riconosciuto da tutti i manager, il quale è responsabile della coordinazione dello stato interno, in caso il leader fallisca può essere eletto un nuovo leader che prende il posto del precedente.

Docker Swarm necessita di almeno un nodo manager per funzionare, per deployment in ambienti di produzione però è consigliato impiegare più nodi manager. In caso di fallimento dei nodi manager i container sui nodi worker restano in esecuzione, ma non possono più essere controllati. Nel dettaglio, in un gruppo di N manager, con N dispari, Swarm tollera la perdita di  $(N-1)/2$  manager senza perdere il controllo dello Swarm. Si consiglia di non superare i 7 nodi manager in uno Swarm. Superati i 7 nodi manager si va incontro a problemi di scalabilità e deperimento delle performance (Docker, n.d.).

I nodi Worker sono istanze Docker Engine che eseguono i container che vengono loro assegnati. Per poter assegnare container a un nodo worker è necessario che sia presente almeno un manager, che nel caso di un deployment su una singola macchina il nodo manager può coesistere con il nodo worker.

## 3 Kubernetes

Annunciato a metà 2014 da Google, Kubernetes è stato donato da Google alla Cloud Native Computer Foundation (CNCF), sviluppata da Google in collaborazione con la Linux Foundation. Kubernetes è nato da Borg, il cluster manager sviluppato internamente a Google, a differenza di Borg scritto in C++, Kubernetes è stato sviluppato in Go. Contrariamente a Docker Swarm, Kubernetes è stato sviluppato principalmente per ambienti cloud-native. Ad oggi è un progetto open source con oltre 115.000 commit su GitHub.

I principali vantaggi di Kubernetes rispetto a Docker Swarm sono la capacità di gestire carichi e cluster di dimensioni maggiori e di disporre di schemi di automazione più granulari e dettagliati. La logica di base di Kubernetes consiste nello sfruttare file o comandi che descrivono le entità che si desiderano avere nel sistema, Kubernetes agirà in modo da realizzare un cluster con le specifiche fornite.

Le principali funzionalità di Kubernetes sono (Kubernetes, n.d.):

- Service Discovery:

I container possono essere esposti tramite un DNS interno oppure usando il loro indirizzo IP, internamente o anche esternamente al cluster.

- Load Balancing:

Anche sfruttando i DNS è possibile distribuire il carico sulle unità di calcolo in base a una serie di parametri e fattori considerati dalle politiche di load balancing.

- Storage Orchestration:

Garantisce differenti modalità per realizzare la persistenza sia in ambienti locali che remoti grazie al supporto di connettori per molti public cloud.

- Rollout e rollback:

Le componenti del sistema possono essere modificate garantendo continuità di servizio, questo è particolarmente rilevante per i container. Nel caso dei container, ad esempio, è possibile aggiornare l'immagine dei container a tutte le istanze in modo graduale e in caso di problemi con il deployment delle nuove immagini è possibile annullare l'aggiornamento, mantenendo i container funzionanti con l'immagine non aggiornata.

- Auto rigenerazione:

Kubernetes può riavviare, terminare o rimpiazzare container che non rispettano determinati standard di salute definiti nei file di configurazione. Qualora i test relativi alla salute dei container fallissero, Kubernetes può agire per riavviare o rimpiazzare automaticamente i container in errore.

- Gestione dei segreti:

La gestione di password, token OAuth e chiavi SSH è facilitata e non obbliga ad inserire informazioni sensibili all'interno delle immagini ma da la possibilità di accedervi a runtime.

### 3.1 Componenti

Un cluster Kubernetes consiste in un insieme di macchine workers, chiamate nodi, che eseguono applicazioni containerizzate, almeno uno di questi nodi deve essere presente in ogni cluster. I nodi worker ospitano i pod, ovvero i componenti dell'applicazione che eseguono codice. Il control plane gestisce i nodi worker e i pod nel cluster. In ambienti di produzione il control plane è composto da differenti macchine e il cluster è supportato da molteplici nodi, al fine di garantire tolleranza ai guasti e alta disponibilità (Kubernetes, n.d.).

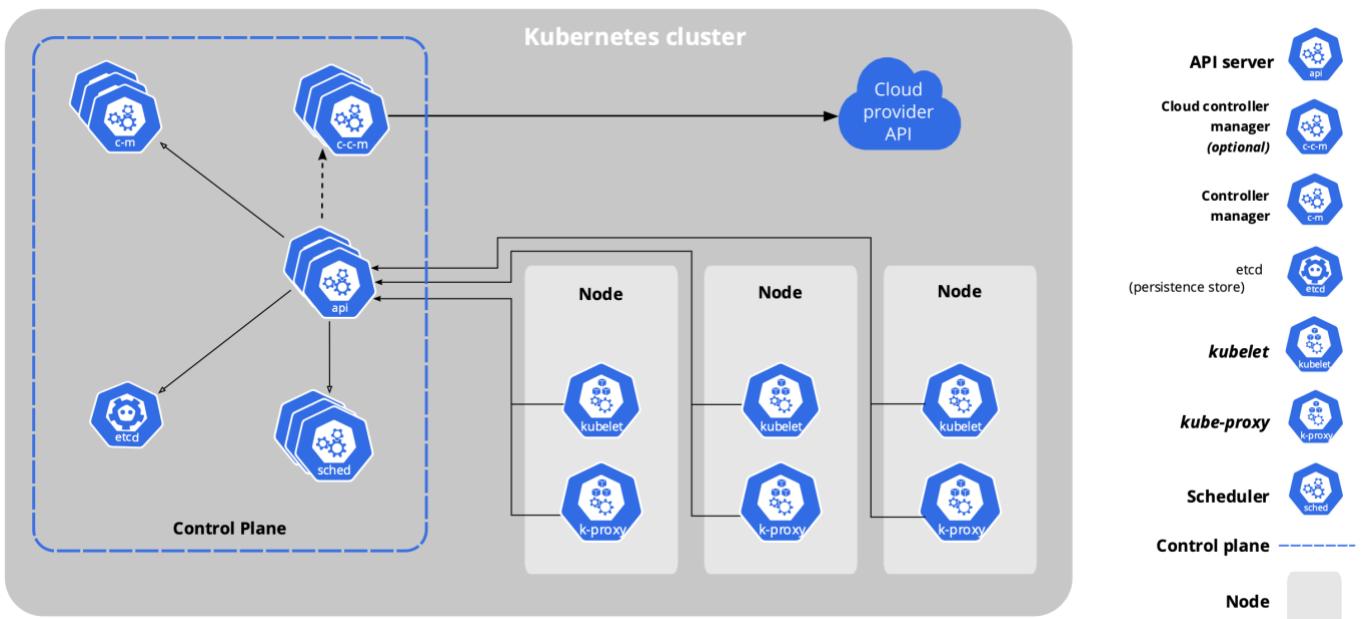


Immagine 13 - Componenti di un cluster Kubernetes

### 3.1.1 Control Plane

Il control plane è a sua volta composto da differenti componenti, che, nel suo complesso collaborano per effettuare tutte le decisioni globali legate al cluster, come decisioni legate allo scheduling o il riconoscimento di pod falliti. I componenti del control plane possono essere in esecuzione su macchine differenti (in caso si abbia un control plane distribuito) ma Kubernetes tende a collocare tutte le componenti del control plane sulla stessa macchina. Sulla macchina scelta per ospitare il control plane di solito non vengono eseguiti container dell'utente, questo per evitare fallimenti all'interno della macchina ed evitare di interferire con il lavoro del control plane.



I componenti del control plane sono:

- Kube-apiserver:

Si tratta punto di accesso al cluster, esistono differenti implementazioni del Kubernetes API Server ma kube-apiserver è la principale. Questa componente si occupa di esporre le funzionalità del cluster tramite REST API. L'apiserver viene utilizzato dagli utenti e da tutti nodi worker per interagire con lo stato condiviso mantenuto dal control plane. Alla luce del grande numero di nodi workers che Kubernetes supporta l'apiserver è stato sviluppato per garantire una grande capacità di scalabilità orizzontale, che ottiene effettuando il deployment di nuove istanze. La scalabilità di questo componente non comporta problemi legati all'accesso a risorse condivise dal momento che si tratta di un componente stateless che si occupa solo di offrire le informazioni presenti nel componente che si occupa della persistenza. Le API possono essere invocate direttamente dall'utente, ma solitamente vengono contattate tramite librerie esterne o tramite kubectl, il tool a linea di comando offerto da Kubernetes per lavorare sui cluster.



- Etcd:

Si occupa di memorizzare permanentemente tutti i dati relativi al cluster e di renderli disponibili tramite un'interfaccia utilizzabile



tramite HTTP e messaggi JSON. Nato dalla CNCF è impiegato in svariati strumenti anche se è principalmente conosciuto per il suo ruolo in Kubernetes. In nome ricorda come l'intento del progetto sia quello di offrire uno strumento capace di compiere gli stessi compiti della cartella /etc nei sistemi Linux, dove la maggior parte delle configurazioni di sistema e delle applicazioni installate risiedono. Nel caso di Kubernetes i dati memorizzati sono configurazioni del cluster (come informazioni sui nodi come IP, porta API e certificati di sicurezza), definizioni di oggetti Kubernetes (come configurazioni dei pod, dei deployment e dei servizi), stato del cluster (come risorse sui nodi, stato dei servizi e stato dei volumi condivisi) e ogni altra informazione che deve essere memorizzata in modo persistente. Etcd consiste in una mappa, fortemente consistente, che ne permette l'utilizzo in scenari distribuiti. Essendo una struttura chiave-valore permette di avere elevata consistenza nei tempi di risposta anche in scenari distribuiti se comparato ad un RDBMS distribuito. La consistenza viene inoltre garantita da una rigida serializzazione degli eventi. Un'altra funzionalità importante consiste nelle change notifications, ovvero la possibilità di notificare i clienti delle modifiche al valore di una chiave specifica o a gruppi di chiavi. Il coordinamento fra i nodi del control plane viene effettuato tramite Raft, il quale definisce il numero massimo di fallimenti tollerati da parte dei nodi del control plane:

Nodi con control plane nel cluster	Maggioranza	Fallimenti tollerati
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3
8	5	3

Tabella 2 - Fault tolerance di nodi con control plane dovuta a etcd

- Kube-scheduler:

Si tratta di un processo eseguito all'interno del control plane che si occupa di assegnare i pod (gruppi di container che collaborano per offrire uno o più servizi) ai nodi worker. Lo scheduler implementa la



logica per individuare il nodo più opportuno su cui avviare un pod. Il processo decisionale che porta alla selezione del Nodo di destinazione di un Pod è diviso in due parti, una iniziale chiamata filtering e una seconda chiamata scoring. Nella parte di filtering lo scheduler genera una lista di nodi che rispettano i requisiti definiti all'interno dei container dentro al pod e all'interno del pod, i criteri presi in considerazione possono essere il numero di processori disponibili o lo spazio in memoria necessario ai container. I nodi selezionati in questa fase sono chiamati feasible nodes, nel caso invece non ci siano feasible node lo scheduler tiene il pod in attesa e non lo schedula sino a che almeno un feasible node sarà disponibile. Durante lo step successivo, ovvero lo scoring, viene stilata una classifica dei feasible node, infine il pod verrà schedulato sul feasible node con lo score più alto. Sia la fase di filtering che quella di scoring utilizzano algoritmi deterministici e integrati nel kube-scheduler, oltre agli algoritmi standard si possono anche selezionare profili di scheduling che impiegano algoritmi alternativi. Nel dettaglio il processo di scheduling segue 12 passaggi chiamati extension points, ognuno di questi passaggi può essere impostato per impiegare logiche personalizzate provenienti dagli scheduling plugins. Esempi di scheduling plugins, impostati di default, sono: ImageLocality che agisce nella fase di score favorendo i nodi su cui stanno già girando pod uguali a quello da schedulare oppure NodePorts agendo in fase di preFilter e filter scartando i nodi che non hanno le porte richieste dai pod libere. Kube-scheduler non è l'unica implementazione di scheduler Kubernetes ma è quella di riferimento e usata di da Kubernetes, tramite altri plugin è possibile sostituire del tutto lo scheduler o modificarne solo alcuni passaggi.

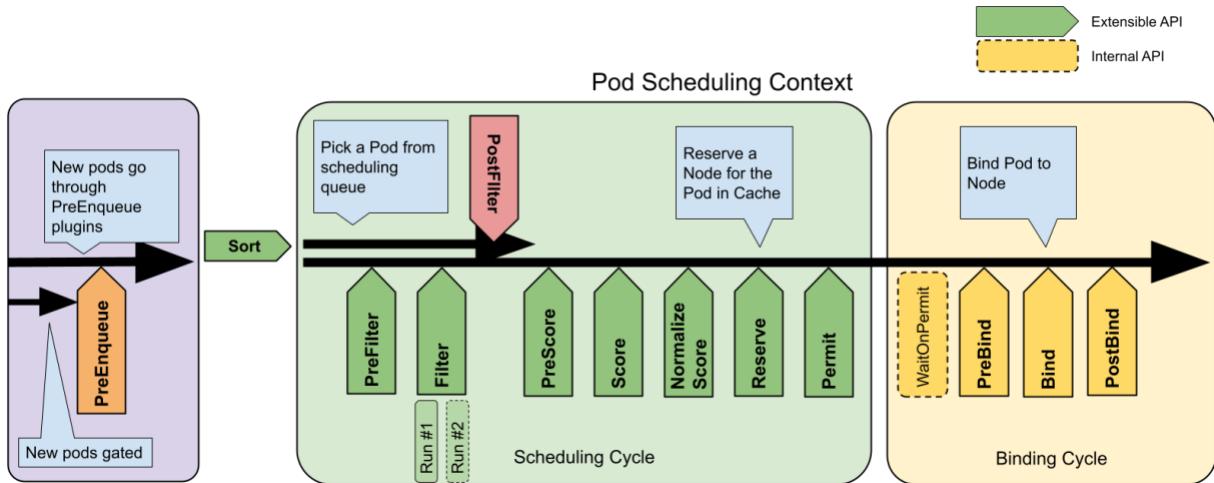


Immagine 14 - Extension point di scheduling (Kubernetes, 2023)

- Kube-controller-manager:

Il kube-controller-manager è costituito da un demone che si occupa di portare e mantenere le risorse presenti nel cluster nello stato definito nei rispettivi file di configurazione. Kubernetes utilizza un approccio dichiarativo, dove l'utente definisce lo stato in cui desidera avere una serie di risorse e Kubernetes si occupa di creare quelle risorse e di mantenerle nello stato richiesto. Il controller è il componente che rende questo controllo possibile. Il kube-controller-manager è in realtà composto da sotto-controller come ad esempio job-controller, deployment-controller e replication-controller, ognuno dei quali si occupa di mantenere lo stato di tipi di risorse differenti. I controller hanno 3 componenti principali: gli informers, i listwatchers e i reconciler. I primi due servono a mantenere il controller cosciente dello stato del cluster, lo stato è ottenuto interfacciandosi con l'api-server che oltre a memorizzare lo stato del cluster su etcd invia notifiche al kube-controller per informarlo di variazioni alle risorse del cluster. Il reconciler invece include la logica che serve a decidere quale azione compiere in conseguenza del cambiamento dello stato di una data risorsa. (Kubernetes, 2022).



- Cloud-controller-manager:

Questo componente non è fondamentale all'utilizzo di Kubernetes in uno scenario on premise ma diventa fondamentale in scenari hybrid o multi-cloud. Il cloud-controller, infatti, si occupa di collegare il cluster a un'infrastruttura cloud, lavorando tramite le API offerte dal cloud provider è in grado di garantire che le risorse sul cloud si trovino nello stato richiesto. Come nel caso del kube-controller anche il cloud-controller è in realtà formato da sotto-controller. Il cloud-controller è strutturato usando un meccanismo a plugin che permette ad ogni cloud provider di offrire una propria versione che si interfacci con le proprie API (Kubernetes, 2023).



### 3.1.2 Nodi Worker

I nodi worker possono essere macchine fisiche o VM collegate al cluster, i worker si occupano di eseguire i compiti definiti dal control plan e



possono interagire fra di loro per condividere informazioni sullo stato dei nodi, dei servizi disponibili sul proprio nodo e per garantire la collaborazione fra servizi in esecuzione all'interno di ogni nodo. Ogni nodo worker all'interno di Kubernetes deve disporre dei seguenti componenti per funzionare correttamente:

- Kubelet:

È la componente principale sui nodi worker, è un processo in esecuzione su ogni nodo worker e si interfaccia con il control plane Kubernetes attraverso l'api-server. Dopo aver ricevuto le specifiche di un pod kubelet garantisce che il pod venga eseguito rispettando quei parametri.

- Kube-proxy:

Funge da proxy per ogni comunicazione sulla rete da e verso i pod. È in grado di gestire l'inoltro di pacchetti TCP, UDP o stream SCTP. Il proxy gestisce anche il traffico che si sposta all'interno del nodo fra pod differenti. Sfrutta il sistema di filtraggio dei pacchetti presente sul sistema della macchina qualora fosse disponibile (iptables ad esempio).

- Container runtime:

Per poter eseguire container è necessario disporre del container runtime. Kubernetes supporta qualsiasi runtime che implementi la Kubernetes CRI (Container Runtime Interface) ma i principali runtime usati sono containerd e CRI-O

### 3.1.3 Addon

Gli addon sono processi che garantiscono supporto al cluster Kubernetes senza essere indispensabili e senza spesso essere parte delle installazioni base. Offrono funzionalità legate principalmente alla connettività, alla service discovery o al monitoraggio.

Il DNS è l'addon principale, in quanto addon non è considerato essenziale e il cluster può funzionare anche senza di esso ma Kubernetes ne consiglia l'utilizzo, spesso infatti lo si trova già installato in molte distribuzioni. Un esempio di addon DNS è Cluster DNS è un server DNS che funge da record per i servizi offerti dal cluster.

Un'altra importante categoria di addon sono i Network Plugin, ovvero componenti che implementano la CNI (Container Network Interface) e sono responsabili di assegnare gli indirizzi IP ai pod permettendogli di comunicare l'uno con l'altro internamente al cluster, i principali sono Calico e Flannel.

Addon di monitoraggio e controllo sono puramente opzionali e non si trovano tendenzialmente nelle installazioni standard, Dashboard è uno di questi e offre un’interfaccia di monitoraggio utilizzabile tramite webapp, Container Resource Monitoring si comporta in modo analogo ma è orientato al log dei container.

### 3.2 Objects

Gli oggetti Kubernetes rappresentano il contenuto del cluster, la loro esistenza e il loro corretto funzionamento sono la ragione per cui il cluster Kubernetes è in esecuzione. Una volta definito un oggetto, Kubernetes si occuperà di mantenere quell’oggetto nel suo stato desiderato, definito anch’esso assieme all’oggetto.

La creazione, la modifica e l’eliminazione di oggetti Kubernetes avviene attraverso le Kubernetes API messe a disposizione dall’api-server sul control plane, o attraverso i vari strumenti che si appoggiano alle API.

Quasi ogni oggetto Kubernetes include due valori che regolano la configurazione dell’oggetto: le specifiche dell’oggetto e lo status dell’oggetto. Le specifiche di un’oggetto devono essere definite prima della sua creazione, fornendo una descrizione delle caratteristiche che si desidera che la risorsa abbia, ovvero il suo stato desiderato. Lo stato descrive invece lo stato presente dell’oggetto, fornito e aggiornato dal sistema Kubernetes e dai suoi componenti. Il control plane di Kubernetes opera per far corrispondere sempre lo stato di un’oggetto alla sua specifica.

Gli oggetti sono rappresentabili in formato JSON ma per ragioni di semplicità e leggibilità vengono scritti in YAML. Dopo l’invio degli oggetti in formato YAML all’api-server si occupa kubectl di convertire il file YAML in JSON per poterlo utilizzare (Kubernetes, 2023).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

*Immagine 15 - File YAML di un oggetto Deployment*

Nell'immagine 15 è mostrato un esempio di file YAML per la creazione di un Deployment Kubernetes. Nel dettaglio ogni file YAML deve possedere dei campi obbligatori, ovvero: apiVersion che indica quale versione di API dell'api-server usare, kind che indica l'oggetto da creare, metadata per identificare l'oggetto con name, UID e opzionalmente un namespace, spec ovvero lo stato desiderato dell'oggetto.

Di seguito un'analisi dei principali tipi Kubernetes:

### 3.2.1 Pod

I pod sono le più piccole unità di computazione di cui può essere eseguito il deployment e che possono essere gestite in Kubernetes (Kubernetes, 2023).

I pod sono gruppi di uno o più container che vengono schedulati ed eseguiti assieme, sullo stesso nodo e condividendo lo stesso contesto. Gruppi di container vengono raggruppati all'interno di un unico pod quando devono essere fortemente dipendenti l'uno dall'altro o quando realizzano

funzionalità logicamente molto correlate. Fra container all'interno dello stesso pod, ad esempio, è molto facile comunicare tramite le porte esposte dei container su localhost.

Spesso in Kubernetes viene usato il modello “one-container-per-pod” che prevede di avere un solo container racchiuso all'interno di un pod. Kubernetes non ha controllo all'interno del pod, per questo motivo quando è necessario scalare un servizio, Kubernetes aumenta il numero di pod e non il numero di container all'interno di un pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: dummy-pod
spec:
  containers:
    - name: node-service
      image: batman.imolab.it:5000/node-service
      env:
        - name: SERVICE_NAME
          value: "Almost welldone Service"
        - name: PROXY_WEBSOCKET_PORT
          value: "80"
        - name: PROXY_ENDPOINT
          value: "localhost"
      ports:
        - containerPort: 8080
    - name: kafka-proxy
      image: batman.imolab.it:5000/kafka-proxy
      env:
        - name: KAFKA_ENDPOINT
          value: "kafka-service.default.svc.cluster.local"
      ports:
        - containerPort: 80
  dnsPolicy: ClusterFirst
```

Immagine 16 - File YAML di un oggetto Pod

Il pod mostrato nell'immagine 16 mostra il file che descrive un pod contenente due container. Il file YAML definisce lo stato ideale del componente, contenuto nel campo specs. Per ogni

container vediamo il nome assegnato al container, l'URL dell'immagine del container, una serie di variabili d'ambiente sotto al campo env e la dichiarazione di quale porta il container deve esporre al resto del pod. In merito alla dnsPolicy, l'esempio usa una policy ClusterFirst che indica al pod di risolvere i nomi prima tramite il DNS interno del cluster e solo successivamente, nel caso fosse necessario, tramite i DNS del nodo su cui risiede il pod.

I file all'interno di un container sono effimeri, questo può creare dei problemi dal momento che i pod non sono costruiti per resistere nel tempo. L'astrazione dei volumi in Kubernetes permette di risolvere il volume della persistenza.

I pod possono infatti disporre di storage, i volumi condivisi disponibili sul pod verranno condivisi da tutti i container del pod. L'approccio che spesso si utilizza è quello di non salvare dati persistenti all'interno dei pod o dei container ma di appoggiarsi a volumi esterni per la persistenza, questo è possibile grazie all'astrazione dei Volumi fornita da Kubernetes.

Kubernetes assegna ad ogni pod un indirizzo IP univoco, all'interno del pod però tutti i container condividono lo stesso spazio, condividendo sia l'indirizzo IP assegnato al pod sia le porte, questo risulta essere un aspetto delicato ed è compito dei container coordinarsi al fine di spartirsi le porte disponibili. I container possono comunicare all'interno del pod usando memoria condivisa POSIX o tradizionali IPC a livello di sistema operativo. L'interazione fra container su diversi pod deve però sempre avvenire tramite indirizzamento IP, sfruttando eventualmente i DNS interni al cluster.

Al fine di monitorare i pod al meglio, Kubernetes definisce una serie all'interno del ciclo di vita dei pod:

- Pending:

Quando un pod è stato accettato dal cluster Kubernetes ma uno o più container non sono ancora pronti all'avvio. I pod si trovano in questo stato quando sono in attesa di essere schedulati o quando l'immagine di uno o più container sta venendo scaricata.

- Running:

Quando il pod si trova su un nodo e tutti i container sono stati creati e pronti a partire, o quando almeno uno dei container sta ancora eseguendo.

- Succeeded:

Quando tutti i container all'interno del pod sono terminati con successo e non necessitano di restart.

- Failed:

Quando tutti i container in un pod sono terminati ma almeno uno è terminato ritornando un codice di uscita diverso da 0 o è stato terminato dal sistema.

- Unknown:

Quando informazioni in merito allo stato dei pod non sono state ottenute, accade solitamente per problemi di comunicazione con il nodo su cui il pod è ospitato

Lo stato dei pod è legato allo stato dei container al suo interno, per questo motivo Kubernetes si occupa di monitorare anche lo stato dei container. Gli stati che un container può assumere sono 3: Waiting (fase di pull dell'immagine), Running e Terminated.

Tramite i pod si possono definire diverse politiche di riavvio dei container in caso di fallimento (o di terminazione) dette restartPolicy. La policy di default è Always, che riavvierà il container ogni qualvolta esso terminerà, OnFailure e Never rispettivamente lo riavvieranno solo in caso di fallimento o non lo riavvieranno mai.

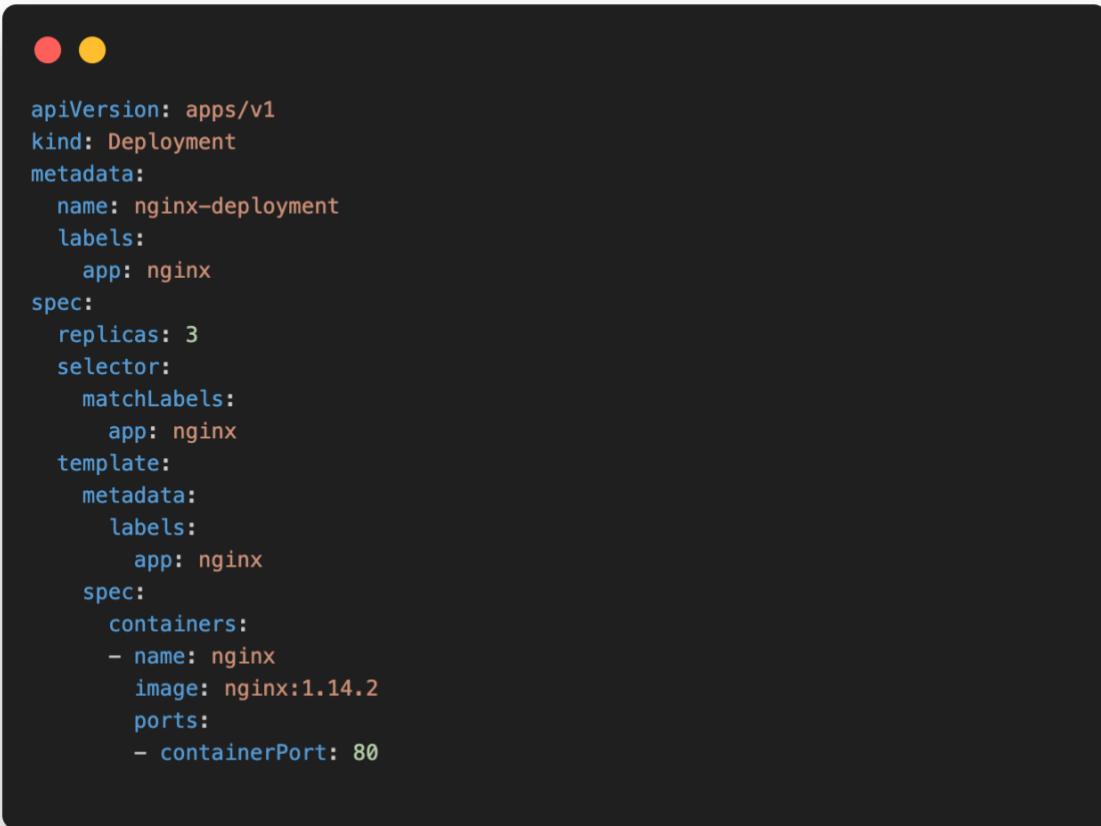
In caso di fallimento Kubernetes impiega un back-off time esponenziale prima di riavviare il container, il tempo di back-off definisce quanto tempo sarà necessario aspettare dalla terminazione del container fino alla sua nuova esecuzione. Il back-off time parte da 10 secondi e arriva fino a 5 minuti, dopo 10 minuti di esecuzione del container invece il back-off viene resettato e in caso di errore si ripartirà da un tempo di 10 secondi.

I pod però raramente vengono eseguiti come oggetti individuali e indipendenti, in Kubernetes i pod sono visti come entità temporanee e non permanenti. Esistono altre sovra-entità che fanno affidamento sui pod per costituire invece un oggetto permanente e duraturo, come i deployment.

### **3.2.2 Deployment**

I deployment sono oggetti Kubernetes volti tendenzialmente a contenere intere applicazioni (o parti rilevanti di esse). Un deployment è composto da uno o più pod, dei quali possono essere definiti il numero di repliche, ovvero quante istanze di un dato pod avere. I deployment sono in grado di gestire il rollout anche graduale di nuove versioni dei container.

Un concetto importante da introdurre è quello dei replicaset ovvero set di istanze dello stesso pod. Modificando il numero di istanze di un replicaset è possibile scalare il numero di pod che il replicaset contiene senza dover avviare manualmente altri Pod. I replicaset sono generalmente gestiti dai deployment Kubernetes e non è quindi necessario gestirli manualmente.

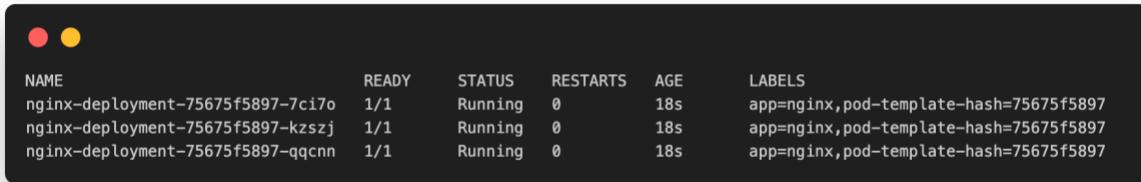


```
● ●
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

Immagine 17 - File YAML di un oggetto Deployment

L’immagine 17 mostra un esempio di file YAML per definire un Deployment. Oltre ai metadati presenti in ogni oggetto Kubernetes, è da evidenziare il campo replicas che definisce la dimensione del replicaset, ovvero il numero di istanze di pod da eseguire. Il resto delle specifiche indica il tipo di pod da eseguire, in modo analogo ai file usati per i pod. Aspetto interessante dei metadati è il campo label, che serve ad etichettate il deployment al fine di creare gruppi di deployment su cui lavorare.

Una volta avviato, il deployment mostrato sopra avvierà 3 pod che verranno eseguiti in modo indipendente l'uno dall'altro:



NAME	READY	STATUS	RESTARTS	AGE	LABELS
nginx-deployment-75675f5897-7c17o	1/1	Running	0	18s	app=nginx, pod-template-hash=75675f5897
nginx-deployment-75675f5897-kzszz	1/1	Running	0	18s	app=nginx, pod-template-hash=75675f5897
nginx-deployment-75675f5897-qqcnn	1/1	Running	0	18s	app=nginx, pod-template-hash=75675f5897

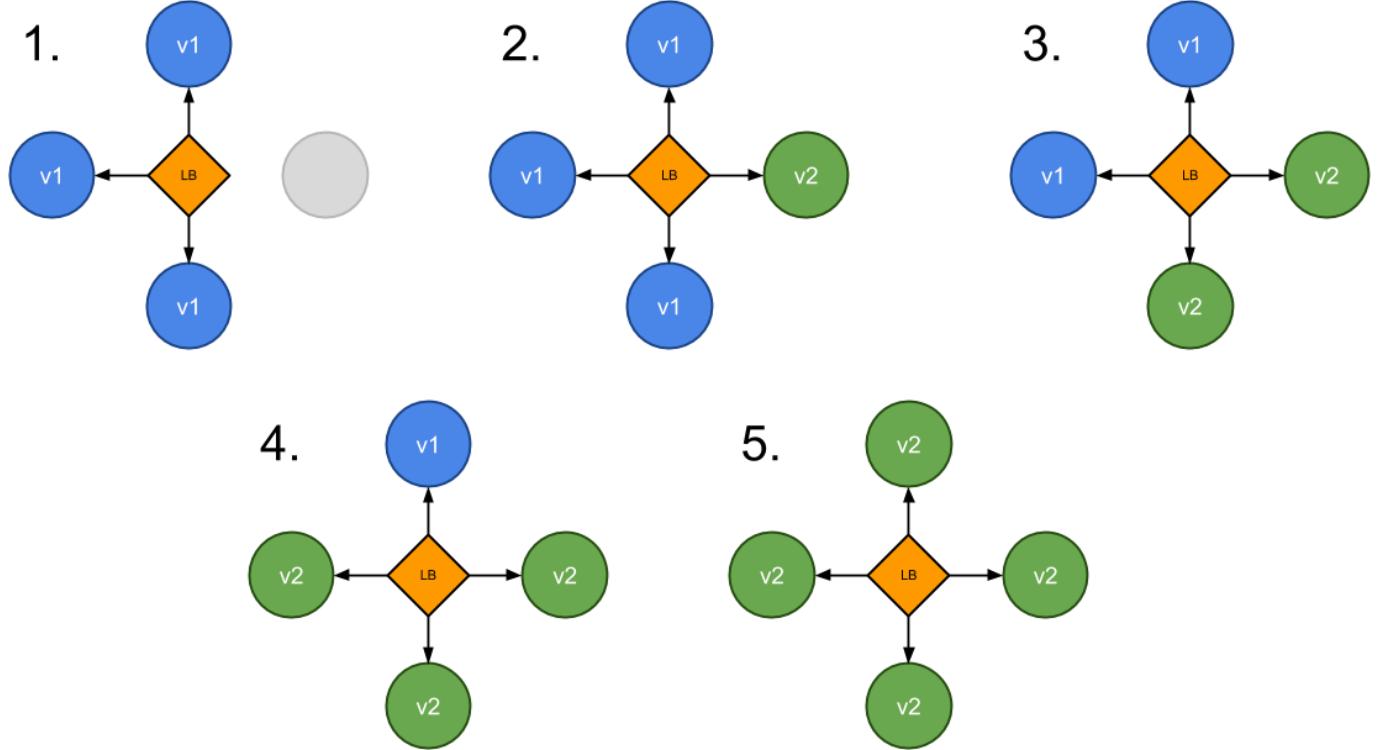
*Immagine 18 - Lista dei pod avviati all'interno di un deployment*

I deployment Kubernetes garantiscono un servizio più affidabile rispetto ai pod, dal momento che un numero maggiore di pod in esecuzione può garantire resistenza ai guasti. I deployment definiscono anche i valori entro cui il numero di pod nel replicaset può variare, di default i deployment accettano un calo del 25% nel numero di pod disponibili e al massimo il 125% dei pod richiesti, questo significa che su un deployment di 4 pod, Kubernetes cercherà di garantire un numero di pod compreso fra 3 e 5. Le oscillazioni nella dimensione dei replicaset avvengono spesso durante gli aggiornamenti dei container (Kubernetes, 2023).

I deployment permettono di aggiornare i pod (o meglio, le immagini dei container che sono eseguiti all'interno dei pod del deployment) in modo graduale e controllato in modo da non avere interruzione del servizio durante gli aggiornamenti che implicano il riavvio dei pod.

Gli aggiornamenti graduali delle immagini dei container, anche chiamati rolling updates, sono molto facilitati da Kubernetes. La modifica del documento YAML del deployment e il suo invio all'api-server con apposito comando è sufficiente per effettuare la modifica di un qualsiasi specifica del deployment. Nel caso di aggiornamenti all'immagine dei pod, Kubernetes crea un nuovo replicaset, dedicato ai nuovi pod, e imposta la sua dimensione a 1 in modo da eseguire un pod aggiornato a fianco dei 3 pod da terminare. Una volta che il pod viene avviato correttamente Kubernetes diminuisce la dimensione del replicaset con i pod non aggiornati, terminando uno di essi e incrementando successivamente il replicaset con i nuovi pod. In questo modo il numero dei pod oscilla sempre fra i 3 e i 4 pod attivi allo stesso momento e non ci sono quindi momenti in cui il sistema non è disponibile.

Durante tutta la fase di aggiornamento un load balancer si occupa di ribilanciare il carico solo sui container correntemente in esecuzione.



*Immagine 19 - Esempio di rolling update [38]*

Qualora ci fossero problemi di stabilità o fosse necessario annullare gli aggiornamenti fatti, è possibile eseguire un rollback. Un rollback consiste nell'annullare le modifiche apportate e nel caso di un deployment consiste nel riportare le immagini dei pod a una versione precedente a quella attuale. Il processo che viene usato per il rollback è il medesimo del rollout, l'unica differenza è che il nuovo replicaset che viene creato contiene pod con una versione di immagine meno aggiornata rispetto a quelli del replicaset correntemente attivi.

Kubernetes permette di utilizzare altri pattern per il rilascio degli aggiornamenti come i Canary Deployment, Blue/Green Deployment o Recreate Deployment.

Oltre alla modifica della versione dei pod è possibile anche modificare il numero di pod attivi, questo può tornare particolarmente comodo nel caso fosse necessario scalare un'applicazione per

far fronte a un carico maggiore o nel caso fosse possibile ridurre il numero di istanze per risparmiare capacità di calcolo e diminuire i costi soprattutto se si è in ambienti cloud.

### 3.2.3 Service

I Service sono un importante livello di astrazione in Kubernetes e permettono di esporre servizi senza lasciar trasparire da chi o in che modo sono implementati. Ogni servizio espone un gruppo di endpoint logici per comunicare attraverso la rete, il componente Service definisce anche la politica che stabilisce in che modo rendere i pod sottostanti accessibili. I servizi si appoggiano a pod vengono selezionati in base alle label inserite nei metadati dei pod stessi o dei deployment che li contengono. Il principale compito dei service è quello di inoltrare le richieste alle porte dei pod, usando l'IP e la porta corrispondente al pod. Durante l'indirizzamento ai pod vengono anche applicate differenti politiche di bilanciamento e gestione del traffico. I servizi vengono identificati dentro un cluster sempre tramite il loro nome logico, il nome logico viene risolto attraverso variabili d'ambiente o più spesso tramite DNS (CoreDNS è il server DNS di default presente in Kubernetes).

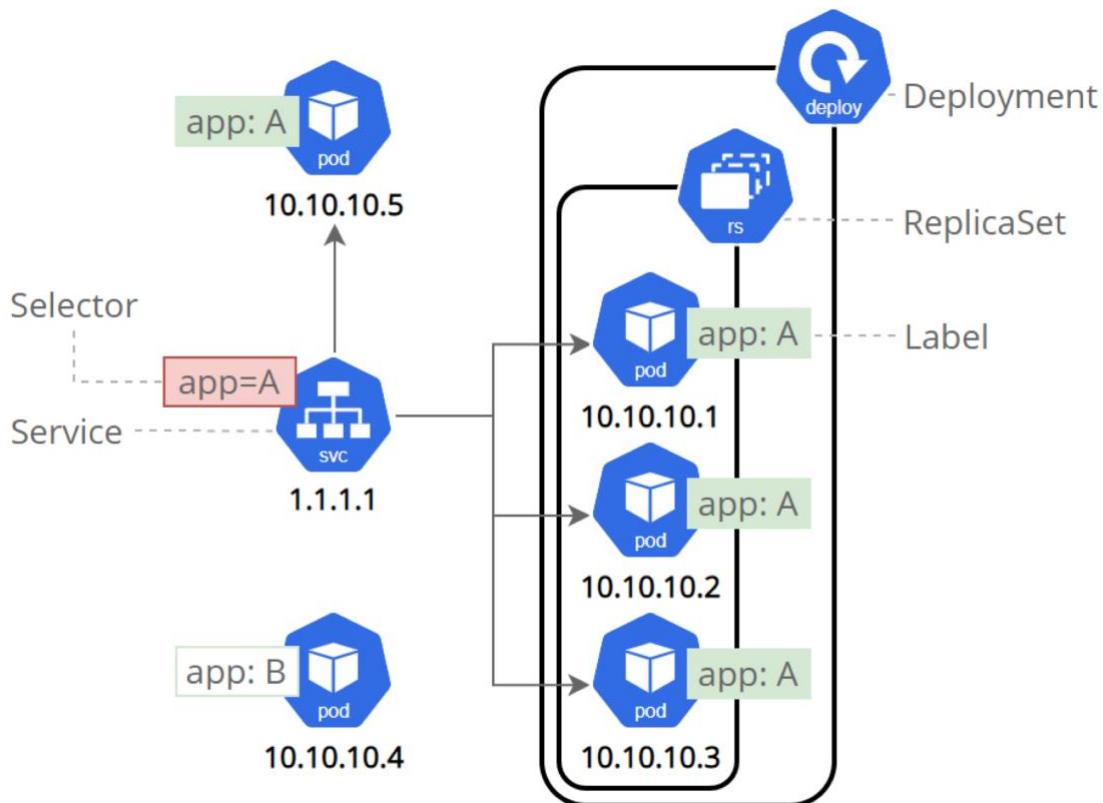


Immagine 20 - Esempio di Service (Kubernetes, 2023)

```
● ●  
apiVersion: v1  
kind: Service  
metadata:  
  name: dummy-node-service  
spec:  
  selector:  
    app: dummy-node-app  
  ports:  
  - name: http  
    protocol: TCP  
    port: 8080  
    targetPort: 8080  
  type: NodePort
```

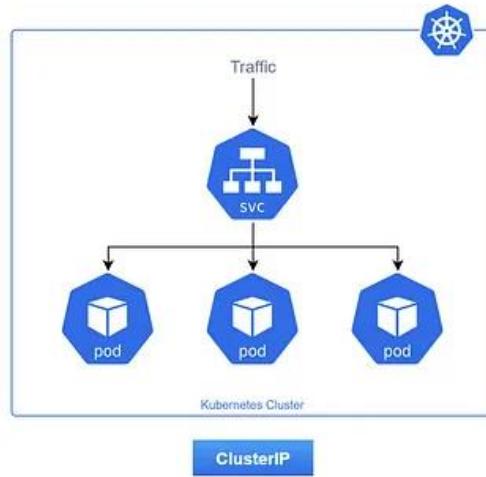
*Immagine 21 - File YAML di un oggetto Service*

Il codice nell'immagine 21 definisce dentro il campo selector il filtro da applicare per trovare i pod a cui appoggiarsi, in questo caso quelli con label app impostata a dummy-node-service.

La sezione ports indica a quali porta fare forwarding, quali porte quindi esporre all'esterno, mentre il type NodePort è uno dei 4 tipi di servizi possibili:

- ClusterIP:

Espone il servizio su un IP interno al cluster, rendendo quindi il servizio raggiungibile solo da dentro al cluster. L'IP assegnato al servizio fa parte di una pool di IP e di default viene assegnato automaticamente, è anche possibile impostare un IP statico. ClusterIP è il tipo di servizio che viene impiegato nel caso in cui non si imposti alcun valore nel campo type.

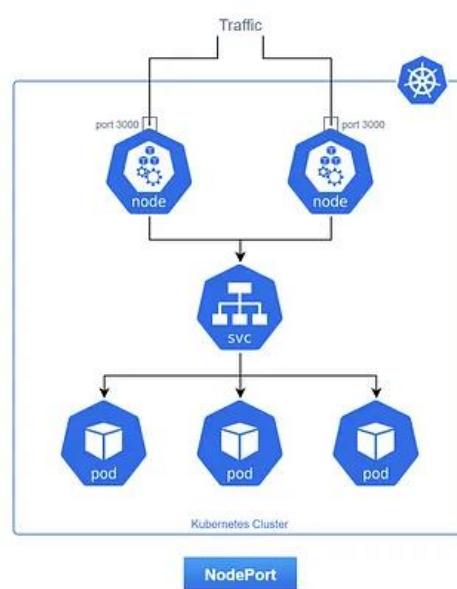


*Immagine 22 - Esempio di Service di tipo ClusterIP (Patel, 2021)*

- **NodePort:**

Esponde il servizio in ogni nodo del cluster su una porta statica e identica per tutti i nodi. Tutte le richieste che giungono ad un qualsiasi nodo sulla porta del servizio vengono automaticamente reindirizzate al servizio, e di conseguenza ai pod. Le porte assegnate al servizio vengono selezionate fra una pool di porte (30000-32767) ma è possibile indicare una porta definita dentro il file di configurazione.

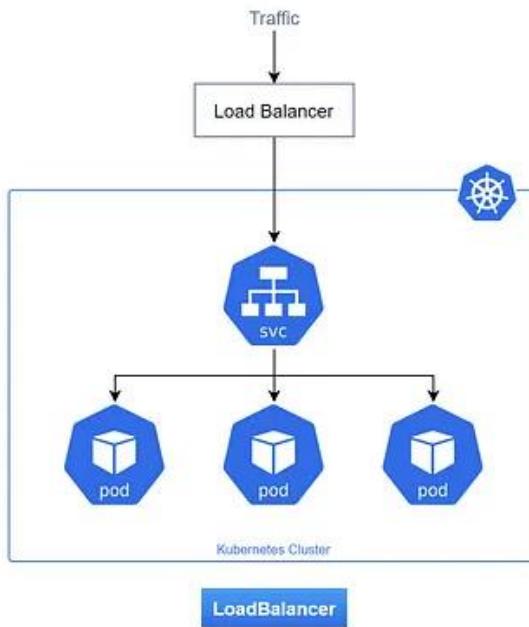
Questo tipo di servizio viene spesso usato quando è necessario implementare servizi di load balancing interni al cluster.



*Immagine 23 - Esempio di Service di tipo NodePort (Patel, 2021)*

- LoadBalancer:

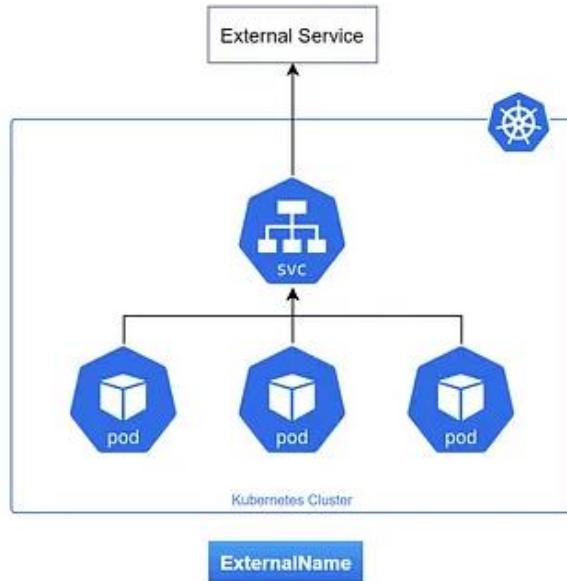
Viene utilizzato soprattutto in scenari cloud in cui è necessario usare un load balancer fornito dal provider. La creazione del load balancer del provider avviene in maniera asincrona e tramite lo stato del servizio è possibile ottenere informazioni sullo stato del load balancer che il servizio usa. Questa logica permette di nascondere la complessità e le differenze fra differenti load balancer offerti da differenti cloud provider.



*Immagine 24 - Esempio di Service di tipo LoadBalancer (Patel, 2021)*

- ExternalName:

Mappa il servizio al contenuto di un externalName, solitamente un URL di servizi esterni. Quando interrogato, il DNS, invece che rispondere con il riferimento a un pod, come nel caso degli altri tipi di servizi, risponde con il riferimento a externalName, tramite un record DNS con CNAME impostato a externalName. Questo approccio dona grandissima flessibilità e permette di fare riferimento a risorse che potrebbero essere disponibili dentro il cluster, quindi mettendo due servizi in serie, oppure risorse disponibili fuori dal cluster.



*Immagine 25 - Esempio di Service di tipo ExternalName (Patel, 2021)*

In conclusione, i service Kubernetes svolgono un ruolo importante nell'orchestrazione e gestione dei servizi su Kubernetes. Forniscono infatti un punto di accesso stabile e indipendente dalle modifiche che avvengono spesso all'interno di deployment e pod. I pod a cui il servizio fa riferimento possono cambiare nodo, essere riavviati, aggiornati o cambiare in numero, ma il servizio si occupa di mantenere un singolo punto di accesso sempre inalterato alla pool dei pod. La flessibilità e modularità dei servizi offerti tramite i service viene notevolmente aumentata senza rendere necessario considerare tutto lo stack di componenti che realizza un servizio come pod o deployment.

### 3.3 Network

Per realizzare l'infrastruttura di rete interna al cluster Kubernetes utilizza un network model, il network model definisce le regole di base: ogni pod ha un suo IP, container dentro lo stesso pod comunicano tramite localhost, i pod possono comunicare fra di loro usando direttamente gli IP, possibilità di definire regole opzionali per ogni pod per restringere o modificare le politiche di utilizzo della rete.

Kubenet è il componente standard di Kubernetes che si occupa di gestire gli aspetti basici della connettività, ma nella maggior parte dei cluster vengono impiegati plugin di terze parti che si integrano con il cluster attraverso le API offerte dalla CNI (Container Network Interface).

All'interno di Kubernetes possono avvenire 4 tipi di collegamenti diversi, si tratta di collegamenti container-to-container, pod-to-pod, pod-to-service e external-to-service.

Nella comunicazione container-to-container, più container dentro lo stesso pod posso riferirsi l'uno all'altro tramite localhost. Per realizzare questo comportamento viene creato un networking namespace e vengono collegati tutti i container allo stesso bridge di rete, così facendo ogni richiesta fatta in localhost viene propagata a tutti i container, anche per questo motivo i container devono coordinarsi per evitare di esporre le stesse porte all'interno del pod.

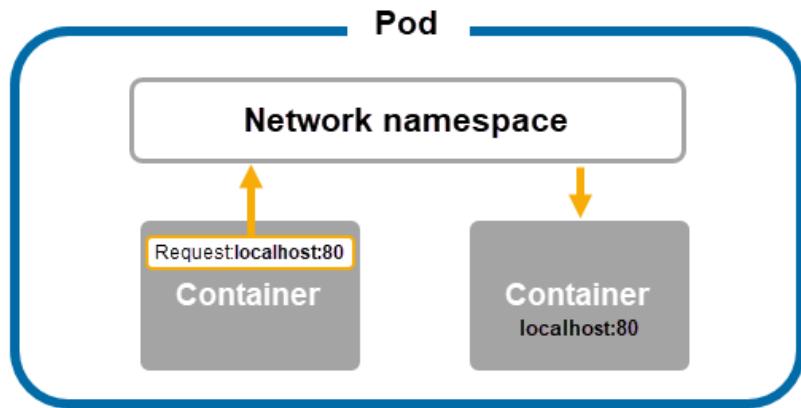
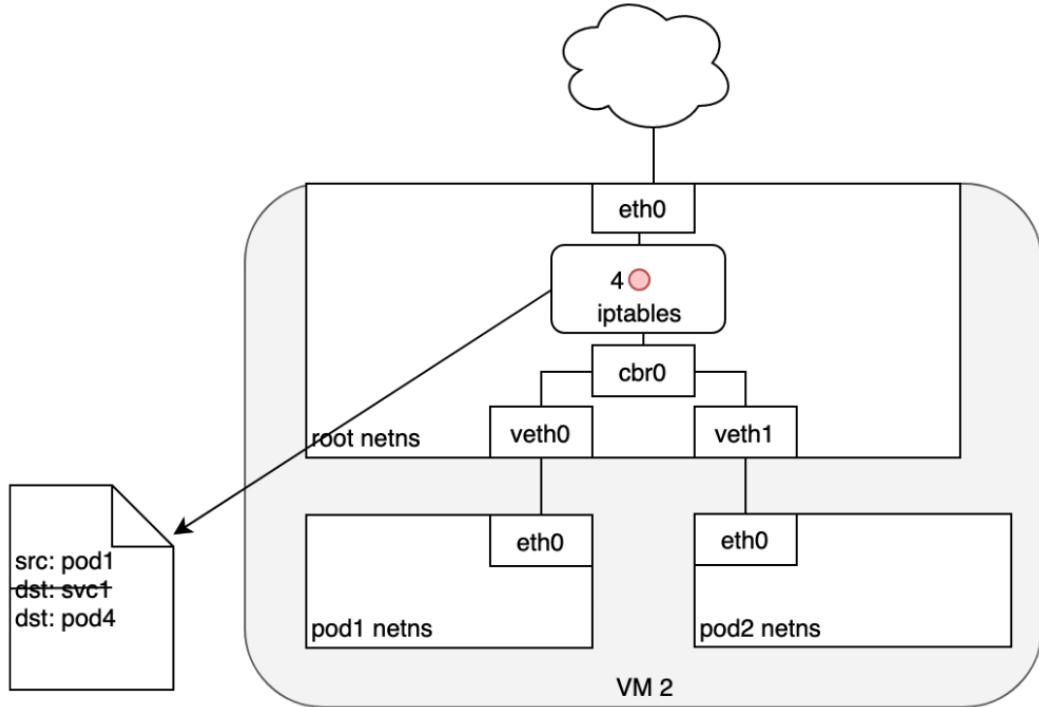


Immagine 26 - Comunicazione container-to-container (Dancuk, 2022)

La comunicazione pod-to-service è invece mediata dai service. I service mappano un singolo IP virtuale e rispettivo hostname immutabile a un gruppo di pod fortemente mutabili.

Il kube-proxy si occupa di generare, gestire e aggiornare le regole iptables su ogni nodo per effettuare le ridirezioni dei messaggi destinati al service. Quando un pacchetto destinato a un servizio arriva bridge del pod (cbr0), non avendo alcuna corrispondenza con l'IP del servizio il bridge inoltra il pacchetto alla rete del cluster tramite l'interfaccia eth0, prima di arrivare all'interfaccia eth0 però tramite le iptables il pacchetto viene sottoposto a un destination natting

(DestinationNAT) per essere rediretto all'IP di un pod che fa parte dei pod mappati a un servizio. La scelta in merito a quale pod usare come destinazione dipende dalle impostazioni del servizio. Dopo il DNAT il pacchetto si comporta come un pacchetto che fluisce verso un pod normalmente.



*Immagine 27 - Comunicazione pod-to-service, passaggi post DNAT (Sookocheff, 2018)*

Le comunicazioni external-to-service si comportano in modo analogo. Con comunicazioni external-to-service si intendono le comunicazioni che si effettuano verso un servizio di tipo ExternalName. In questo caso si impiega una risoluzione DNS per redirigere le richieste a un servizio esterno al cluster.

La comunicazione pod-to-pod si basa su una serie di interfacce virtuali e su un bridge interno al nodo che funge da tabella di instradamento. Nel dettaglio ogni nodo ha una connessione virtuale (veth1, veth2, vethx...) che si collega in tunnelling alla rispettiva interfaccia sul pod (eth1, eth2, ethx...).

Il nodo possiede un'interfaccia cbr0, che collega a un bridge a cui sono connesse tutte le interfacce virtuali. Il bridge funge da tabella di routing per capire a quale interfaccia virtuale inoltrare ogni pacchetto ricevuto in base all'IP di destinazione.

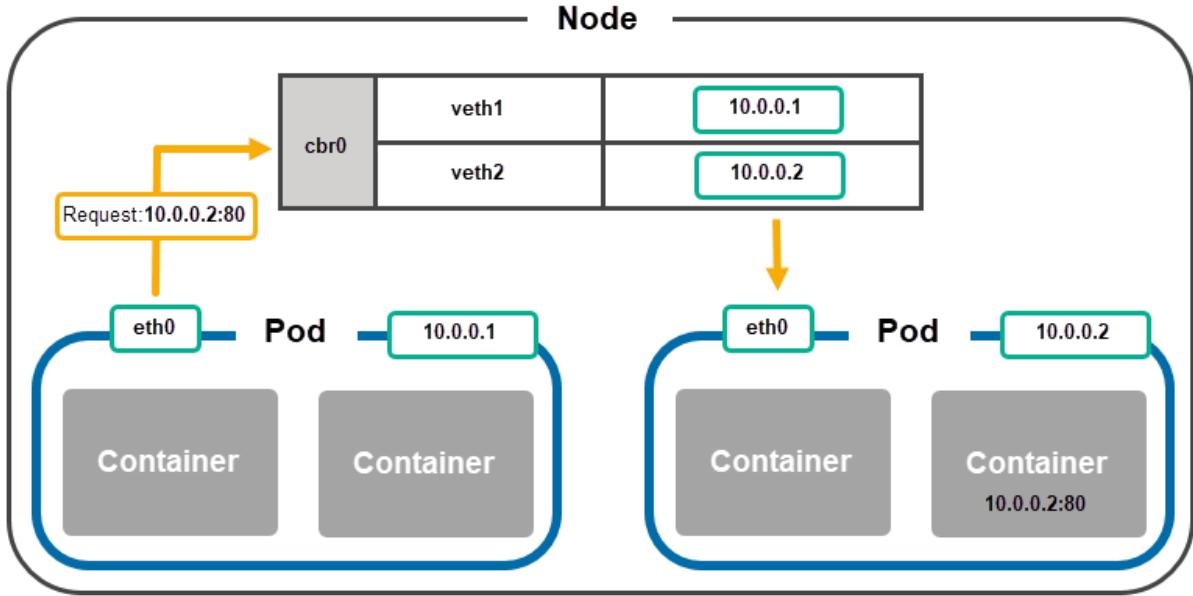


Immagine 28 - Comunicazione pod-to-pod (Dancuk, 2022)

Nel caso in cui il bridge cbr0 non sia in grado di trovare un riferimento nella sua tabella, esso inoltra la richiesta alla tabella di routing del cluster, la quale opera con la stessa logica ma invece che referenziare IP appartenenti a pod, lavora con le subnet. In questo ultimo caso si parla di node-to-node communication.

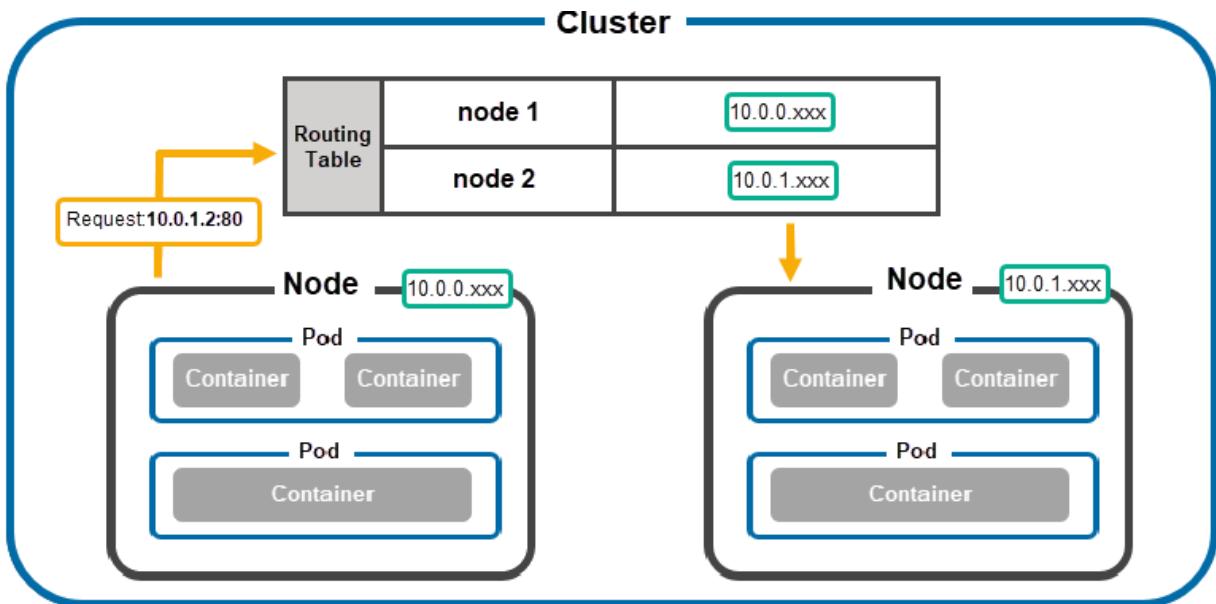


Immagine 29 - Comunicazione node-to-node (Dancuk, 2022)

### 3.4 Storage

Dal momento che i container e i pod in Kubernetes sono effimeri, quindi con un ciclo e una durata di vita non del tutto controllabile, è importante avere la possibilità di aggiungere uno stato esterno ai pod.

La persistenza in Kubernetes si basa sui volumi, esistono vari tipi di volumi le cui caratteristiche cambiano in base al tipo. Tutti i volumi sono viste come directory che possono essere accedute dai container nei pod. I volumi vengono montati nel filesystem dei container definendo quale volume montare e il percorso all'interno del filesystem del container in cui montarlo.

Un tipo di volume fondamentale è il PersistentVolume (PV), ovvero un oggetto creato tramite YAML dentro il cluster Kubernetes, che fa riferimento ad un servizio di storage di qualsiasi tipo, on-premise o on-cloud (Azure Bucket, Azure Disk, NFS, ISCSI...). Un PV viene istanziato a livello di cluster Kubernetes ed è poi necessario per ogni pod che intende sfruttarlo effettuare una richiesta di utilizzo. Le richieste di utilizzo di un volume sono chiamate PersistentVolumeClaim (PVC), e definiscono i requisiti che deve avere la risorsa che il pod ha intenzione di acquisire (in termini di spazio, tipo di volume, modalità di accesso...). All'atto della creazione del pod il cluster si occuperà di assegnare un PV o una sezione di esso al pod, in questo modo il pod non è consapevole di come sia implementato il volume che impiega, ma viene garantito che i requisiti richiesti per quel volume siano rispettati. In caso il pod dovesse terminare, il suo sostituto si ricollegherebbe allo stesso PV tramite la stessa PVC.

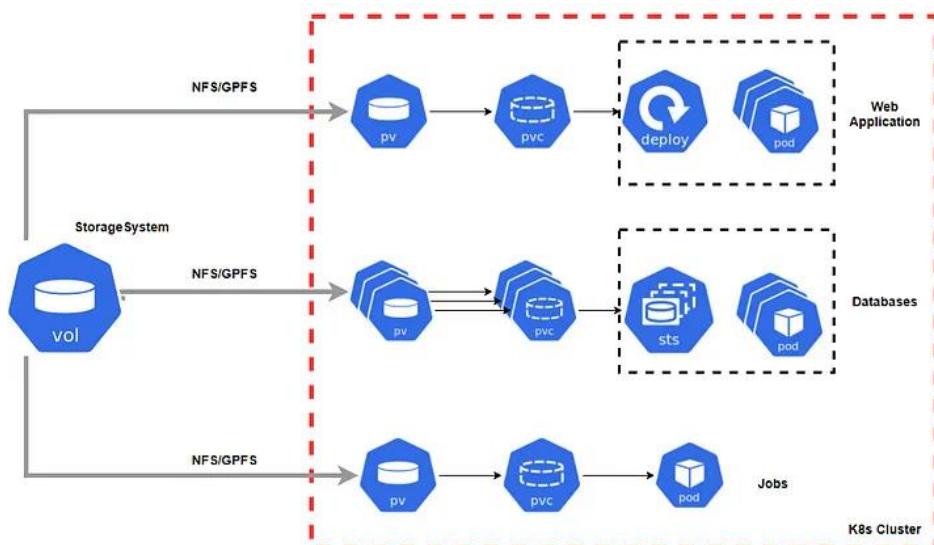


Immagine 30 - Utilizzo di PV e PVC

Questa divisione delle interfacce a due livelli, generata da PV e PVC, è particolarmente utile anche da un punto di vista organizzativo dal momento che il creatore degli oggetti PV sarà un amministratore del cluster mentre il creatore dei PVC sarà l'utilizzatore del cluster.

Altri volumi interessanti sono i ConfigMap e gli EmptyDir, entrambi questi componenti non necessitano di una PVC.

EmptyDir è probabilmente il tipo di volume più semplice, si tratta di storage locale e consiste in una directory montata all'interno del filesystem dell'host che ospita il pod che la utilizza. Ha lo stesso tempo di vita del pod e può essere usato da uno o più container all'interno di uno stesso pod. È solitamente usta per scambiare file e informazioni fra i container di uno stesso pod.

ConfigMap è un tipo di storage in senso lato che viene passato come file all'interno di un pod. Le ConfigMap contengono una serie di coppie chiave-valore che vengono solitamente usate per impostare parametri attraverso variabili d'ambiente all'interno dei container.

Sebbene il fatto di avere due componenti diverse (PV e PVC), gestite e create da attori diversi all'interno di una organizzazione può essere utile, è necessario considerare scenari in cui è necessario creare molte applicazioni e di conseguenza necessitare di molteplici PV con caratteristiche diverse. Al fine di semplificare la creazione e l'utilizzo di volumi si usano le StorageClass, ovvero classi di servizio, o profili di servizio.

Tramite la StorageClass è possibile utilizzare PV istanziati dinamicamente a partire dalle specifiche definite nelle StorageClass, offrendo così un sistema di provisioning di PV dinamico.

Per un utente che desidera ottenere un nuovo PV sarà appena sufficiente indicare il nome della StorageClass all'interno della PVC per ottenere il rispettivo PV.

StorageClass definisce anche una serie di parametri importanti per evitare un uso non corretto dello storage fra cui le ReclaimPolicy, allowVolumeExpansion o le volumeBindingMode.

In conclusione, lo storage in Kubernetes rappresenta un elemento cruciale per garantire la persistenza dei dati e consentire alle applicazioni di funzionare con uno stato. I diversi tipi di volumi offerti da Kubernetes, come PersistentVolume, EmptyDir e ConfigMap, offrono soluzioni flessibili per soddisfare le esigenze di archiviazione delle applicazioni. La separazione tra PersistentVolume e PersistentVolumeClaim fornisce un'interfaccia a due livelli, consentendo agli

amministratori di creare e gestire i volumi di archiviazione, e agli utilizzatori di richiedere i volumi necessari in base alle proprie necessità. L'uso delle StorageClass semplifica ulteriormente la creazione e l'utilizzo dei volumi, consentendo un provisioning dinamico e personalizzato. La corretta gestione dello storage in Kubernetes è essenziale per garantire l'integrità dei dati, la scalabilità delle applicazioni e la resistenza del sistema nel suo complesso.

## 4 ServiceMesh e EventMesh

Il coordinamento fra microservizi comporta un significativo overhead e spesso richiede l'aggiunta di codice non relativo alla logica applicativa dentro al microservizio: l'autenticazione fra le connessioni, routing del traffico, funzionalità di log o load balancing fra servizi e politiche per compensare i fallimenti di rete. Tutte queste funzionalità non sono strettamente legate alla logica di business che dovrebbe essere l'unica realizzata all'interno di un microservizio. Inoltre, molte di queste funzionalità sono le medesime e vanno implementate nello stesso modo in tutti i microservizi appartenenti alla stessa applicazione.

Una delle soluzioni per risolvere o attenuare questi problemi è inserire un livello intermedio fra microservizi e rete per facilitarne le interazioni. ServiceMesh e EventMesh forniscono un'infrastruttura aggiunta che si occupa proprio di questo.

ServiceMesh, nata prima di EventMesh, è diventata necessaria nel momento in cui le applicazioni hanno iniziato a contenere un numero significativo di microservizi, portando ad avere molte interazioni complesse fra microservizi. Inizialmente, prima della nascita di Service ed EventMesh, alcune importanti aziende del settore informatico impiegavano grandi librerie per gestire il coordinamento fra microservizi. Alcuni esempi sono Hystrix di Netflix, Finagle di Twitter o Stubby di Google, tutte e tre queste librerie avevano caratteristiche comuni fra cui essere di considerevoli dimensioni, essere scritte in pochi linguaggi, difficilmente traducibili e molto difficili da distribuire e aggiornare (MacKinnon, 2019). Senza ServiceMesh tutti i microservizi sono obbligati ad implementare la medesima logica per permettere l'interconnessione fra microservizi, invalidando così, almeno in parte, il principio di single responsibility dei microservizi.

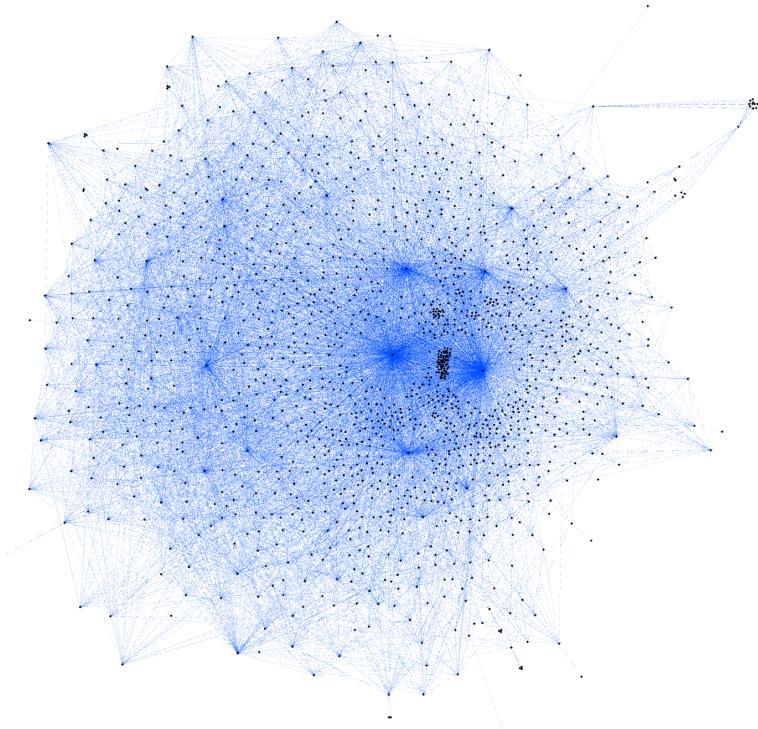
Nonostante la somiglianza tra i termini, event mesh e service mesh sono due infrastrutture distinte. L'event mesh è asincrona mentre la service mesh supporta un tradizionale sistema di messaggistica sincrono di tipo request-reply. Event mesh e la service mesh rappresentano approcci di comunicazione complementari, forniscono un servizio diverso ma comunque efficace per gli sviluppatori, che possono scegliere la soluzione migliore per le applicazioni che stanno progettando.

## **4.1 Definizione ServiceMesh**

ServiceMesh è un livello di infrastruttura configurabile che rende la comunicazione fra applicazioni a microservizi possibile, strutturata e osservabile. Una ServiceMesh monitora e gestisce il traffico di rete fra microservizi, ridirigendolo, garantendolo tramite meccanismi di controllo o limitando l'accesso alle risorse in base alle necessità di ottimizzazione e protezione del sistema (VMWare, n.d.).

Si tratta di spostare il coordinamento e l'implementazione del comportamento legati all'interazione fra servizi, dai microservizi a un'entità esterna all'applicazione stessa ovvero la ServiceMesh. Questo approccio diviene fondamentale quando le connessioni e le interazioni fra microservizi diventano molte e particolarmente complesse.

L'obiettivo di ServiceMesh è di realizzare funzionalità legate alla connettività per i microservizi tramite il suo livello infrastrutturale, ma senza intaccare la logica e la composizione dei microservizi che connette. La strategia che è stata usata per mantenere disaccoppiati il livello infrastrutturale e i singoli microservizi è la logica del proxy. Questo approccio consiste nell'utilizzare un terzo componente chiamato proxy, posto fra microservizio e rete, che ha il compito di garantire, nel modo più trasparente possibile, la realizzazione degli aspetti legati alla connettività per il microservizio a cui è associato.



*Immagine 31 - Architettura a microservizi di Monzo, ogni punto un microservizio, ogni riga una connessione (Kleeman, 2019)*

I principali vantaggi di una ServiceMesh sono:

- Osservabilità:  
È più facile raccogliere e aggregare metriche di utilizzo per ogni microservizio.
- Controllo del traffico:  
Tramite un routing più oculato è possibile aumentare la resistenza ai guasti effettuando ritrasmissioni o facendo riferimento a servizi di fallback.
- Sicurezza:  
Facilita l'implementazione, la gestione e l'impiego di protocolli di comunicazione cifrata come mTLS (mutual TLS) gestendo i certificati, policy di sicurezza e di accesso.
- Uniformità:  
Controllando in modo centralizzato le politiche di accesso e di gestione delle comunicazioni, tramite proxy agnostici è possibile regolare l'accesso alla rete di ogni servizio con gli stessi componenti e le stesse modalità.

I benefici di ServiceMesh risiedono nell'avere un livello aggiuntivo che permette di controllare l'accesso e l'interazione fra i microservizi, il fatto di avere sidecar proxy (un container aggiuntivo dentro ogni gruppo di container per supportare le operazioni legate alla connettività), inoltre porta con sé i benefici del sidecar pattern rendendo una ServiceMesh applicabile ad ogni microservizio e infrastruttura.

Questa complessità e separazione delle responsabilità ha un costo significativo, questi alcuni dei principali svantaggi di una ServiceMesh:

- Consumo di risorse:

Un proxy in ogni pod e le componenti necessarie al funzionamento di ServiceMesh consumano risorse, il consumo però cresce linearmente al numero di servizi.

- Complessità:

Aggiungendo un proxy per pod e componenti per il coordinamento, i potenziali punti di rottura aumentano e aumentano anche le componenti da configurare e far interagire. Sarà inoltre necessario testare non solo le funzionalità delle componenti aggiunte ma anche la loro interazione.

- Debug:

Aggiungere livelli di gestione può rendere difficile capire la sorgente dei problemi e rende più lungo e complesso il debug.

- Latenza:

Aggiungendo passaggi intermedi nella comunicazione la latenza aumenta. Non si tratta solo di passaggi aggiuntivi a livello di rete ma anche passaggi in termini decisionali, necessari per rispettare le regole definite dalla ServiceMesh.

La maggior parte di questi problemi nascono dall'impiego del sidecar proxy, un componente fondamentale per ServiceMesh. Questo componente va istanziato in ogni pod che realizza un microservizio e per questo motivo si parla di consumo di risorse, latenza e complessità aggiuntiva significativi. Avendo un solo proxy per pod i costi in termini di risorse crescono linearmente con il numero di pod, non si tratta quindi di difficoltà ingestibili. È poi da sottolineare che in scenari particolarmente ricchi di microservizi, ServiceMesh risulta essere un approccio necessario e non uno alternativo.

## 4.2 Componenti ServiceMesh

A livello base, un mesh di servizi è costituito da servizi, sidecar proxy e componenti di monitoraggio e coordinamento del sistema. Tutte le richieste da o verso un servizio passano attraverso due proxy all'interno della mesh: il proxy per il servizio chiamante e il proxy per il servizio ricevente. Il piano di controllo (control plane) è l'autorità che fornisce criteri e configurazione al piano dati (Google Cloud, 2021). Il data plane è il componente che implementa le direttive ed è composto dall'insieme dei sidecar proxy.

### 4.2.1 Sidecar

Con sidecar pattern si intende un pattern a singolo nodo composto da due container. Il primo è l'application container e contiene la logica della applicazione, senza di questo la applicazione non esisterebbe. Il secondo è il sidecar container, il cui ruolo è aumentare e migliorare l'application container spesso senza che l'application container ne sia a conoscenza (nel senso che non si comporta in modo strutturalmente diverso dal caso in cui sia da solo). I due container devono essere nello stesso container group, che nel caso di Kubernetes è il pod (O'Reilly, n.d.).

Nel caso specifico di ServiceMesh, il sidecar proxy viene usato per implementare funzionalità legate alla connettività e al rapporto con altri servizi. I sidecar proxy in un deployment Kubernetes vengono implementati aggiungendo un container proxy ad ogni pod e facendo in modo che il sidecar proxy gestisca il traffico entrante e uscente al microservizio.

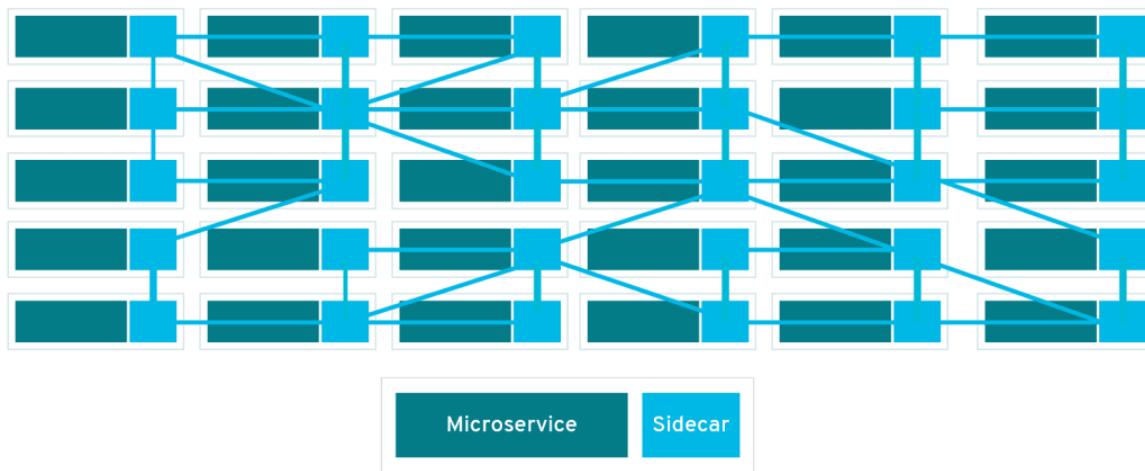


Immagine 32 - Architettura sidecar proxy (Donston-Miller, 2021)

Ogni microservizio ed il proprio proxy comunicano via rete, nel dettaglio di Kubernetes possono comunicare tramite localhost trovandosi sullo stesso Pod, per quanto riguarda invece il protocollo della comunicazione quest'ultimo dipende dal proxy, da come viene implementato e quali protocolli supporta.

Il sidecar proxy funge da intermediario per tutte le comunicazioni in entrata e in uscita dal microservizio e agisce in modo trasparente dal punto di vista del microservizio.

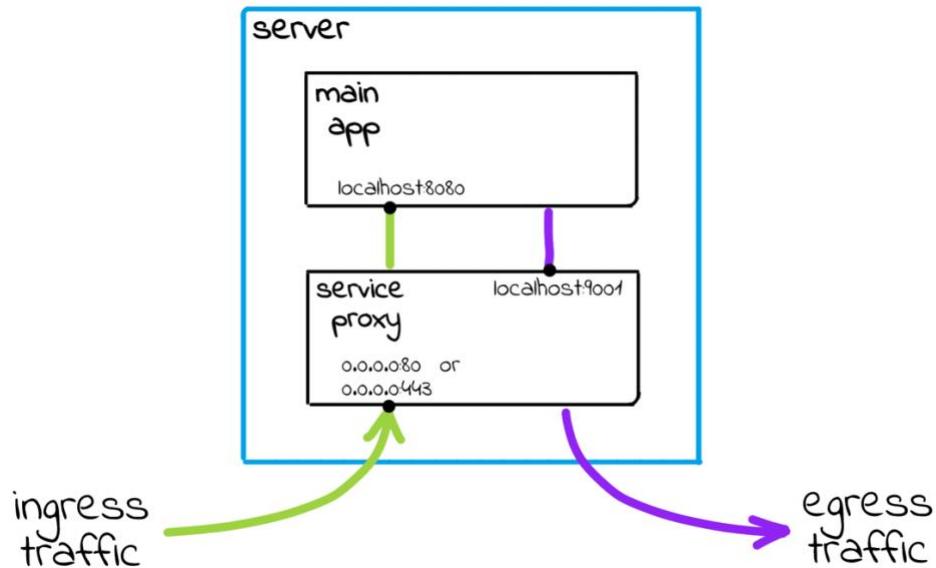


Immagine 33 - Comunicazione fra container e sidecar proxy (Velichko, 2021)

La comunicazione tramite rete offre un importante vantaggio in termini di flessibilità e di modularità dal momento che rende il linguaggio in cui è realizzato il servizio non rilevante. Senza l'impiego di sidecar proxy sarebbe necessario implementare le funzionalità di comunicazione nello stesso linguaggio del microservizio, ma comunicando tramite la rete ed essendo un'entità a parte, il proxy può usare un qualsiasi altro linguaggio. Il fatto di avere un'entità separata che si occupa di svolgere i compiti legati al coordinamento dei servizi permette quindi di evitare di riscrivere il codice e aggiunge gradi di libertà ad entrambe le componenti.

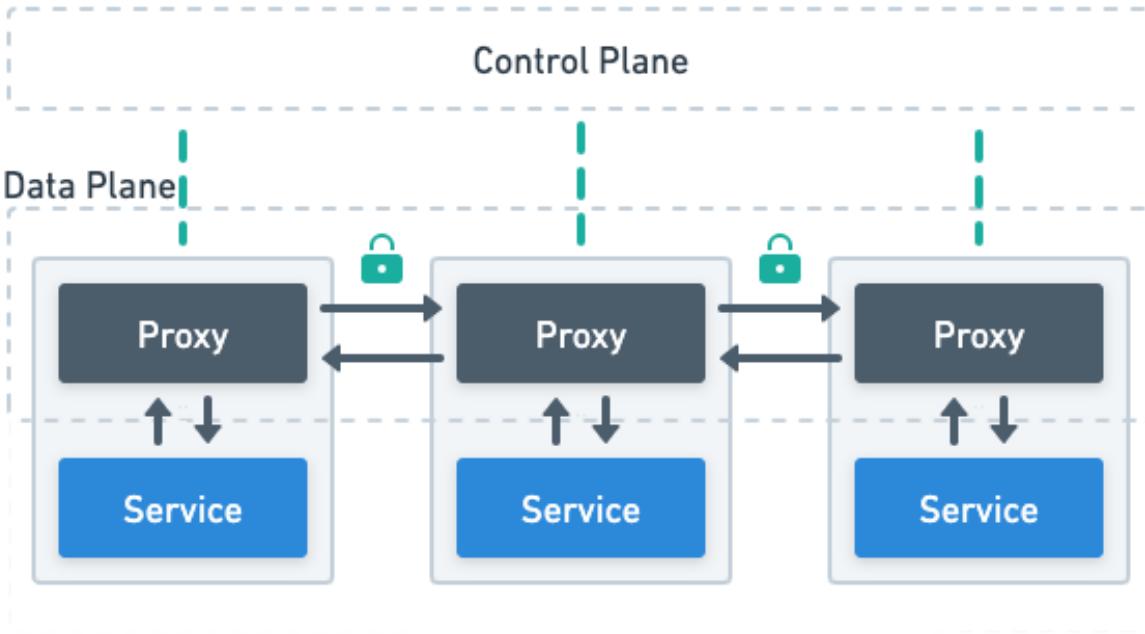
Un'architettura di questo tipo rende particolarmente semplice effettuare monitoraggio dei microservizi, soprattutto se si tratta di un monitoraggio in merito al traffico in ingresso e in uscita. Il monitoraggio di informazioni relative agli application container, in termini di uso delle risorse o eventuali errori, risulta più complesso e non può essere effettuato in maniera del tutto trasparente.

#### **4.2.2 Control Plane**

Il control plane è il componente che all'interno di una ServiceMesh assume il ruolo di coordinatore, lavorando a un livello out-of-band quindi senza mai toccare o interagire direttamente con il traffico di rete. Tramite il control plane è possibile impostare le politiche e le regole che devono essere rispettate dai sidecar proxy e di conseguenza risulta essere il punto di contatto fra amministratore del cluster e i microservizi. Le configurazioni applicate ai sidecar proxy dal control plane riguardano: regole di routing del traffico, politiche di load balancing, politiche volte a garantire la fault tolerance e aspetti relativi all'osservabilità dei microservizi. Il control plane come centro di controllo è necessario siccome i sidecar proxy sono entità effimere in quanto container, è necessario che sia quindi il control plane a fornire uno stato.

#### **4.2.3 Data Plane**

Il data plane, anche conosciuto come proxying layer, è composto dal set di tutti i sidecar proxy appartenenti a tutti i microservizi dell'applicazione. Il data plane intercetta tutti i pacchetti nelle richieste ed è responsabile di realizzare controlli sullo stato di salute del microservizio, routing, load balancing, autorizzazione e generare segnali osservabili per condividere metriche e log. Questo componente realizza solo le direttive fornite dal Control Plane, in nessun caso, e per nessuna funzionalità prende decisioni autonomamente. Il data plane è responsabile sia della comunicazione intracluster sia della comunicazione entrante e uscente dal cluster (O'Reilly, n.d.).



*Immagine 34 - Architettura ServiceMesh (Braun, 2022)*

### 4.3 Implementazioni ServiceMesh

Esistono molteplici implementazioni di ServiceMesh, quasi tutte piuttosto recenti, questo è dato dal fatto che la necessità di una ServiceMesh sorge qualora ci si trovi di fronte ad un grosso applicativo implementato attraverso un alto numero di microservizi, servizi di questo tipo non erano presenti né tantomeno comuni decenni fa. Per questo motivo l'impiego di ServiceMesh è indissolubilmente legato all'adozione dei microservizi e di orchestratori come Kubernetes.

Alcune delle implementazioni più significative sono Istio (Istio, n.d.), AWS App Mesh (AWS, n.d.), Linkerd e Consul .

AWS App Mesh è il servizio di ServiceMesh offerto da AWS, disponibile al pubblico dal marzo 2019, offre un'ottima integrazione con l'ambiente AWS e gli altri servizi offerti come Amazon ECS o Amazon EKS. Punti forti di App Mesh sono la facilità di utilizzo e l'integrazione con altri servizi come AWS CloudWatch (AWS, n.d.) nel caso del monitoraggio. Il limite di App Mesh è però legato agli orchestratori e agli ambienti che supporta: sistemi solo cloud based e ospitati da AWS.

Linkerd è un progetto open source avviato dall'azienda Buoyant e ora parte della CNCF. Linkerd è particolarmente apprezzato per la sua leggerezza, facilità di utilizzo e deployment, ottima anche

l'integrazione con il mondo Kubernetes. La leggerezza e la semplicità di utilizzo sono però a discapito delle funzionalità rispetto ai competitor e una community non particolarmente ampia.

Istio risulta essere l'implementazione open source più completa in termini di funzionalità realizzate e in termini di documentazione disponibile. Per questi motivi è stata considerata l'implementazione più opportuna da analizzare più nel dettaglio.

#### 4.4 Istio ServiceMesh

Istio è la principale implementazione di ServiceMesh, oltre ad essere estremamente completa, supportata e usata, è anche la base per altre implementazioni come Openshift (RedHat, n.d.), la ServiceMesh di RedHat.

Istio nasce nel 2017 dalla collaborazione fra Google, IBM e Lyft, che dopo averlo sviluppato lo hanno donato alla CNCF per rendere il progetto open source e per proseguirne lo sviluppo in questa modalità.

I principali vantaggi di Istio sono le numerose funzionalità e la loro completezza come load balancing, routing avanzato, gestione delle versioni, controllo granulare della sicurezza e crittografia del traffico. Grazie alla forte integrazione con Prometheus, Istio fornisce anche ottime capacità in termini di osservabilità e monitoraggio.

Gli svantaggi invece sono una curva di apprendimento ripida rispetto alle altre implementazioni, non solo nell'utilizzo ma anche nella configurazione iniziale.

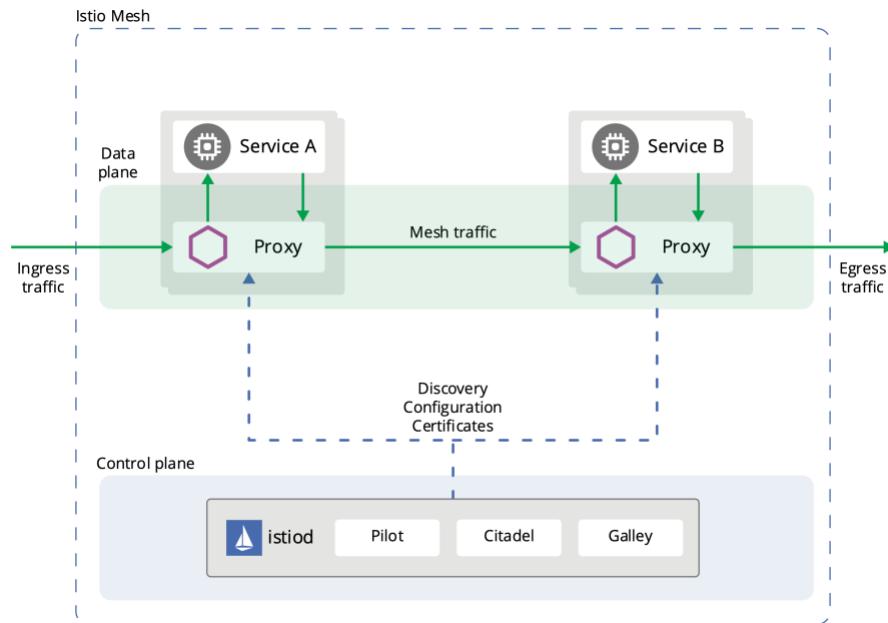


Immagine 35 - Architettura Istio (Istio, n.d.)

#### **4.4.1 Data Plane**

Il data plane è implementato attraverso Envoy proxy, un tipo di sidecar proxy sviluppato da Lyft e integrato in Istio. Si tratta di un proxy ad alte performance, scritto in C++ che, come di norma nelle ServiceMesh, è l'unico componente a gestire direttamente il traffico nella rete. Le principali funzionalità degli Envoy proxy sono: ricerca dinamica dei servizi, load balancing, comunicazione TLS, supporto a HTTP2, gRPC, TCP e WebSocket, circuit breaking in caso uno dei servizi con cui si sta comunicando sia fallito, controlli sulla salute del microservizio, rollout graduale verso servizi fornitori e metriche per il log.

In fase di inizializzazione viene effettuato il processo di sidecar injection, tramite init-container, un container che ha il compito di modificare le regole di routing del microservizio davanti al quale bisogna porre il proxy, questo comporta modifica delle iptables e delle configurazioni di rete per impostare il container Envoy come gateway per tutte le comunicazioni. I proxy Envoy hanno poi la possibilità di essere dinamicamente configurati tramite delle API che espongono e che vengono impiegate dal control plane per modificarne a runtime le impostazioni e il comportamento.

#### **4.4.2 Control plane**

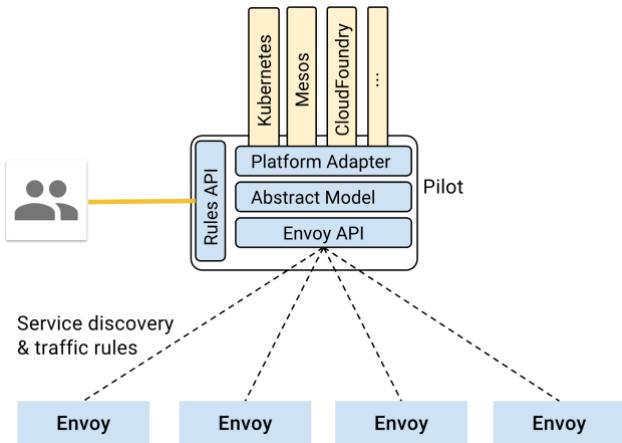
Il control plane è racchiuso in un binario unico chiamato istiod. Inizialmente il control plane era composto da una serie di microservizi ma in seguito ai commenti degli utenti è stato collassato in un unico binario per permettere un'installazione e configurazione semplificata. In termini di funzionalità offerte non ci sono state variazioni.

Il primo componente del Control Plane è il Pilot.

Pilot si occupa degli aspetti relativi alla rete e alla service discovery.

Per quanto riguarda la rete, Pilot raccoglie tramite Rules API una serie di direttive macroscopiche sulla configurazione richiesta e le traduce in direttive che possono essere distribuite ad ogni singolo Envoy proxy. Dopo aver generato le direttive per i singoli Envoy proxy, sulla base della piattaforma sulla quale ci troviamo, si occupa di distribuirle contattando le Envoy API esposte da

ogni proxy per modificarne le configurazioni e il comportamento a runtime. Spesso si usano configurazioni statiche quindi questa fase di setup avviene solo all'avvio e non durante il lavoro. Le funzionalità realizzabili attraverso questo componente sui proxy sono: routing del traffico dinamico, routing per A/B testing, rollout graduale o canary release, failure recovery tramite timeout, retries e circuit breakers e in conclusione fault injection per verificare la compatibilità delle politiche di recupero della rete (Istio, n.d.).



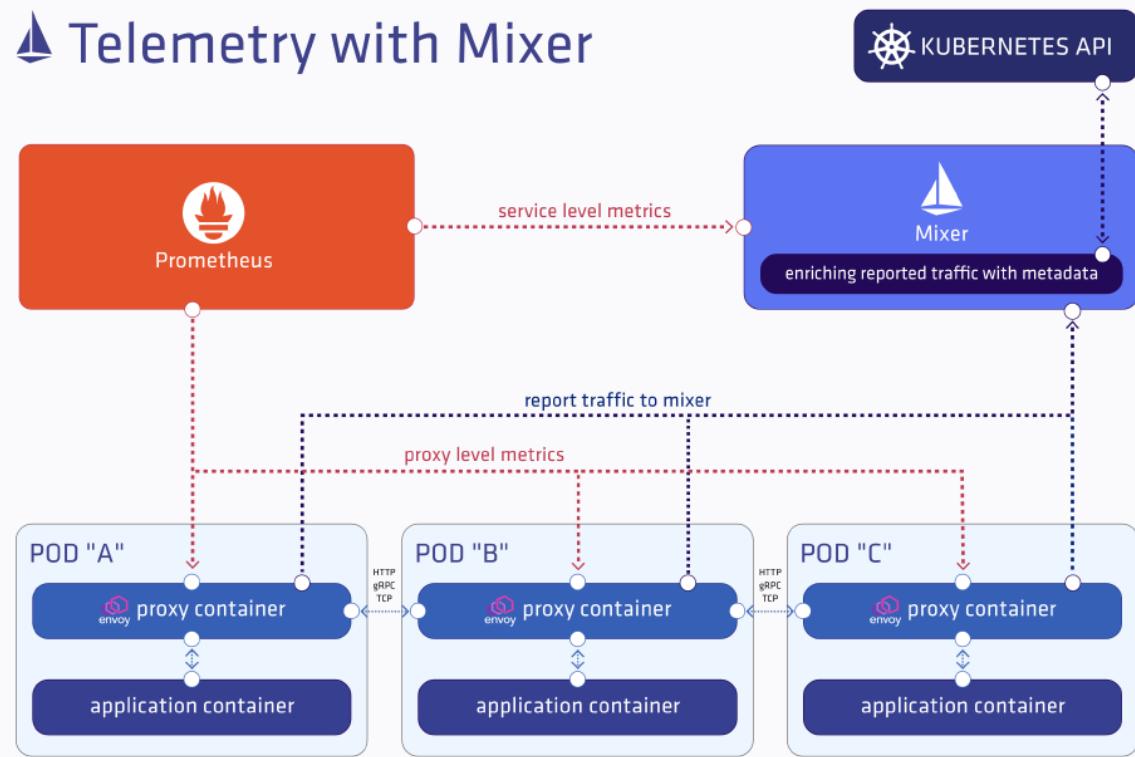
*Immagine 36 - Componente Pilot dell'Istio Control Plane (Istio, n.d.)*

Galley è il secondo componente del control plane. Galley effettua la validazione delle configurazioni, le processa e le distribuisce. È responsabile di isolare il resto del control plane dai dettagli dell'ottenimento dei dati dalla piattaforma sottostante, come ad esempio Kubernetes (Istio, n.d.).

Citadel è il componente che si occupa di tutti gli aspetti legati alla sicurezza della mesh. Permette una forte autenticazione fra servizi e utente finale tramite la gestione delle credenziali e dei certificati necessari. Citadel può anche definire chi può avere accesso a determinati servizi (Istio, n.d.).

L'ultimo componente dell'architettura è Mixer, non fa parte del control plane ma ha un ruolo di rilievo per tutto il cluster in quanto si occupa di raccogliere dati telemetrici sullo stato dei proxy in collaborazione con Prometheus. Nel dettaglio al Mixer arrivano dati per ogni chiamata entrante e uscente che viene fatta dai proxy, sarà poi Mixer ad arricchire queste informazioni aggiungendo informazioni provenienti direttamente da Kubernetes tramite le Kubernetes API. Una connessione

aperta e spesso usata da parte di ogni proxy nel sistema rappresenta un carico molto significativo e diventa molto dispendioso in termini di risorse quando consideriamo molti microservizi.



*Immagine 37 - Prima versione del sistema di log, con Mixer (Varga, 2020)*

Per questo motivo nel 2021 Istio ha rimpiazzato il Mixer con Istio Telemetry V2, un sistema che ha significativamente ridotto l'impatto soprattutto in termini di utilizzo di CPU. Telemetry V2 diventa ancora più isolato dal control plane e le principali funzionalità di raccolta dati precedentemente presenti in Mixer sono state spostate all'interno degli Envoy Proxy tramite plugin. Questa modifica rende più complesso apportare modifiche agli Envoy proxy siccome sono servizi monolitici, ma rende il tutto più snello dal momento che ora la comunicazione è solo fra proxy e Prometheus, parte delle informazioni che Mixer aggiungeva tramite le Kubernetes API vengono ora raccolte dagli Envoy proxy.

## Telemetry V2

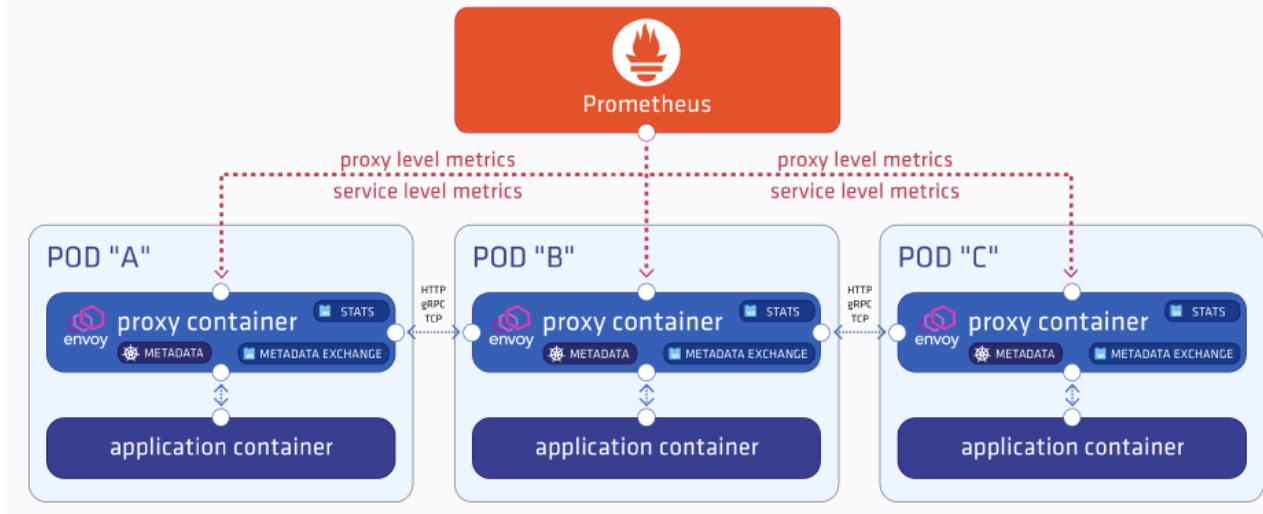


Immagine 38 - Seconda versione del sistema di log, senza Mixer (Varga, 2020)

### 4.4.3 Modelli di deployment

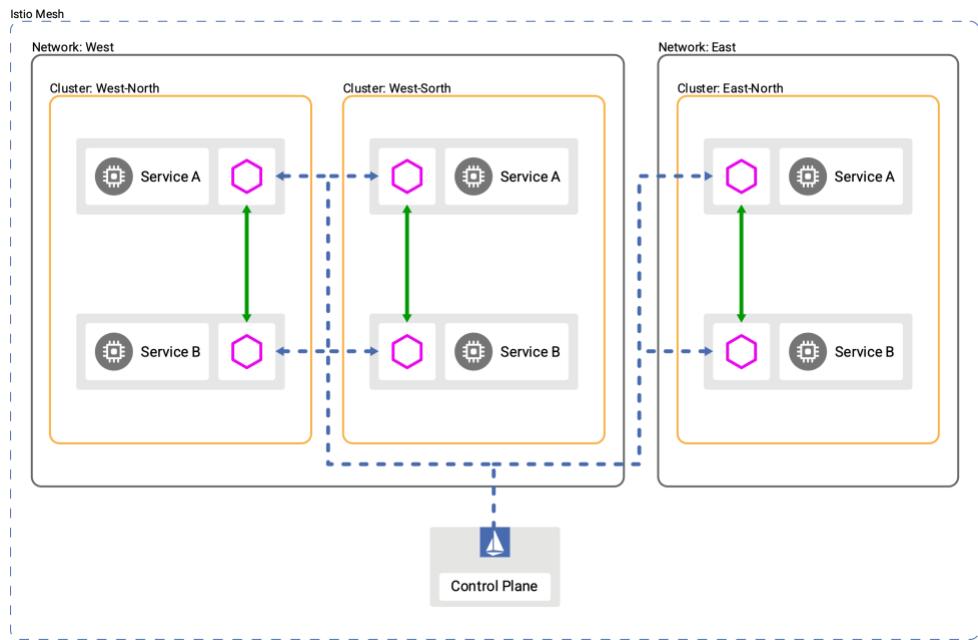
Dal momento che ServiceMesh è impiegato per controllare grandi numeri di servizi è bene definire varie modalità di utilizzo e differenti modalità di deployment sono necessarie in differenti scenari. Nel dettaglio Istio offre diverse opzioni in merito a: numero di cluster, numero di reti, numero di control plane, numero di mesh.

Più cluster possono essere usati tramite Istio, questo può essere utile per garantire isolamento dei servizi, alta disponibilità, performance e avere cluster in differenti posizioni geografiche, per migliorare la latenza o per rispettare regolamenti come la GDPR.

Kubernetes non permette il controllo di molteplici cluster da un singolo punto di accesso, sarebbe necessario operare sui molteplici cluster individualmente per modificare tutte configurazioni.

Isto invece permette di creare un'unica service mesh, con un unico punto di accesso per la modifica e la configurazione della mesh distribuita.

Un deployment multicloud è facilmente implementabile avendo un singolo Control Plane in contatto con tutti gli Envoy proxy su ogni microservizio, indipendentemente dal cluster in cui essi si trovano.



*Immagine 39 - Architettura multicloud (Istio, n.d.)*

Istio può essere usato in scenari in cui si opera su reti diverse, ovvero in reti in cui non tutti i microservizi hanno accesso diretto l'uno all'altro. Molti sistemi di produzione necessitano di subnet per isolare i sistemi e garantire alta disponibilità.

Istio offre supporto per lavorare su più reti, una configurazione di questo tipo è detta multi-network.

Per realizzare una mesh che attraversa più reti è necessario impiegare gli Istio Gateways, ovvero gateway che fungono da nodo di connessione fra le reti. La soluzione è quella di esporre tutti i servizi rilevanti attraverso il gateway. Quasi sempre un deployment multi-network è obbligatorio quando si lavora in multi-cluster.

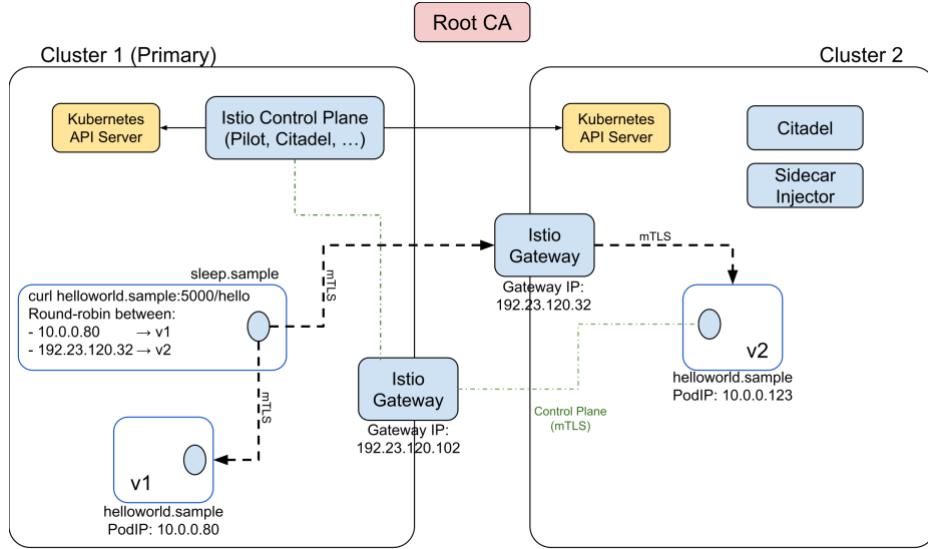


Immagine 40 - Architettura multi-cluster e multi-network (Istio, n.d.)

Il control plane rappresenta una delle componenti fondamentali e può essere organizzato secondo differenti architetture rispetto ai cluster in cui si trovano i data plane.

I 4 principali modelli di control plane deployment sono: primary cluster, primary e remote cluster, external control plane, multiple control planes.

Primary cluster consiste in un deployment di un singolo cluster con control plane incluso.

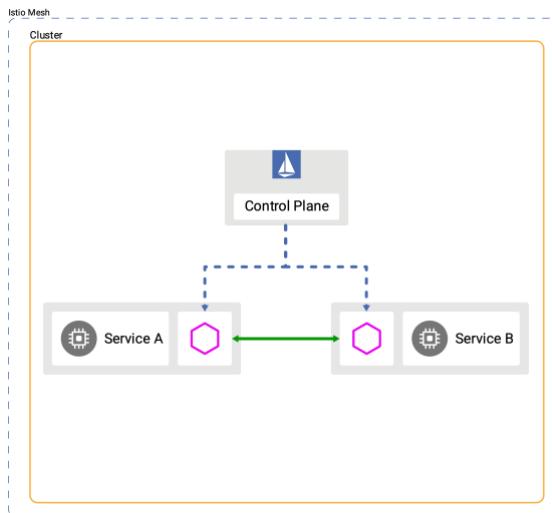
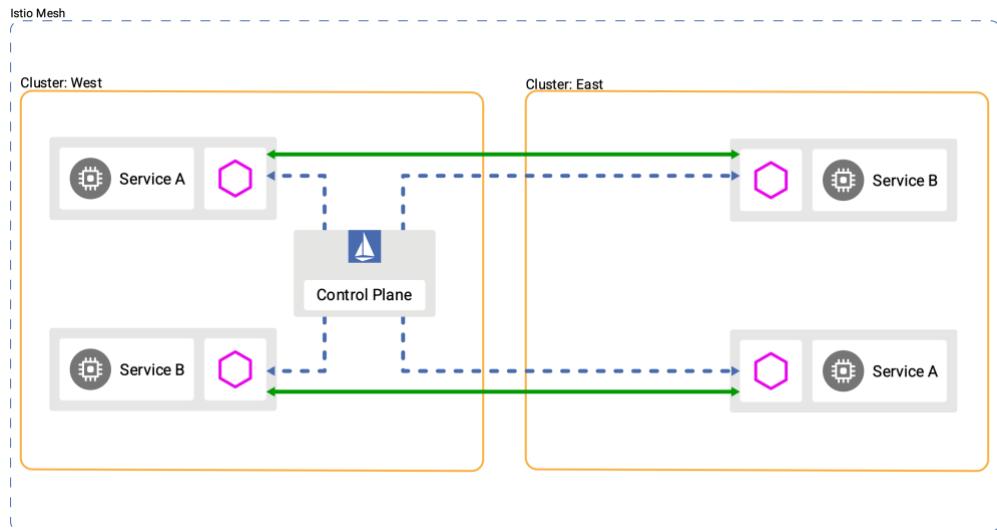


Immagine 41 - Deployment single cluster (Istio, n.d.)

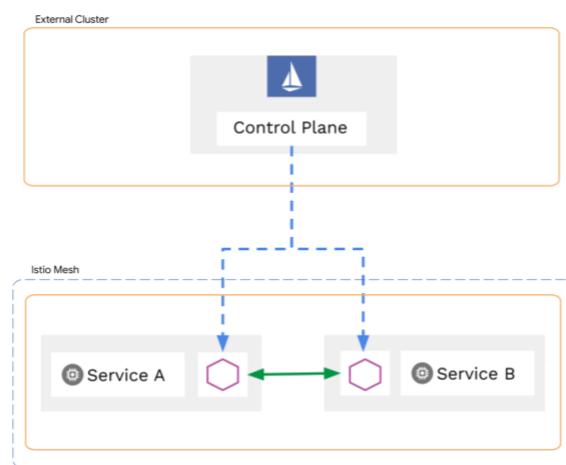
Primary e remote cluster consiste in un primary cluster e uno o più external cluster secondari che non contengono un control plane proprio ma si appoggiano al control plane contenuto nel cluster

primario. È solitamente usato per architetture multicluster e spesso si impiegano Istio gateway per collegare il control plane con gli external cluster.



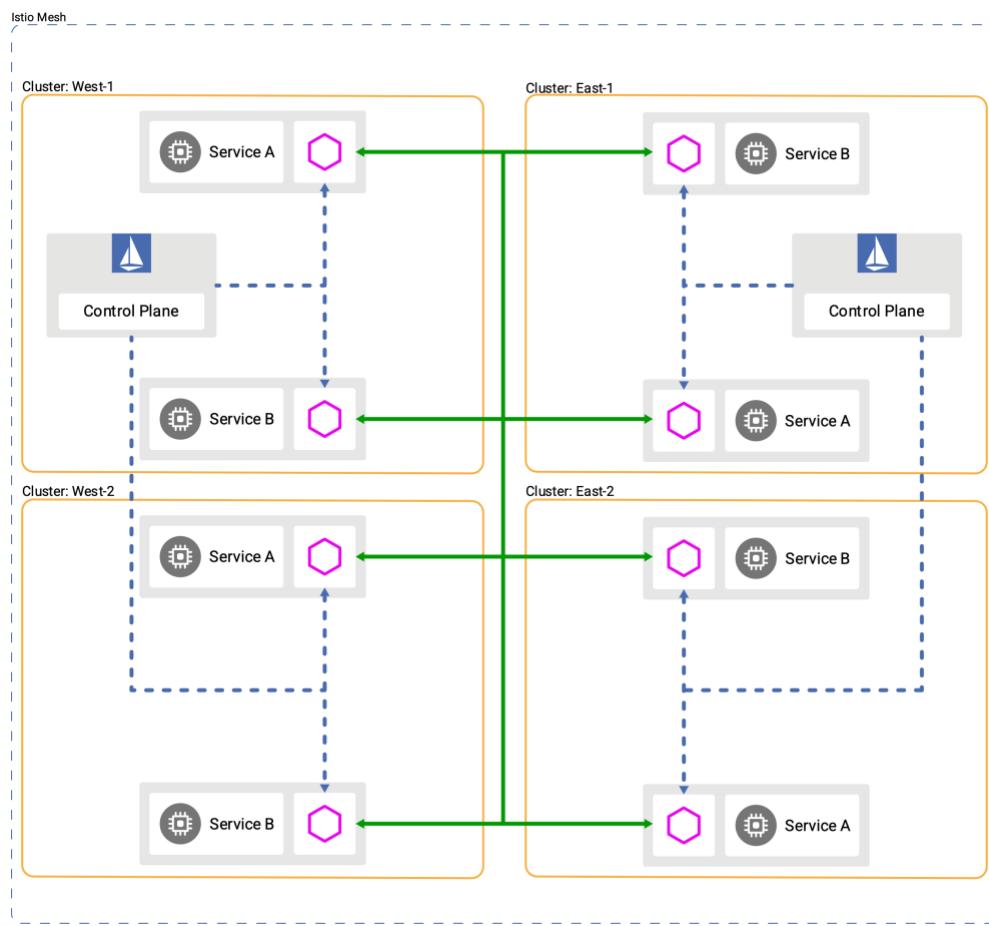
*Immagine 42 - Deployment remote cluster (Istio, n.d.)*

External control plane è il tipo di deployment solitamente usato quando si lavora con un control plane managed. Comporta l'utilizzo di due cluster, uno in cui è contenuto solo il control plane e un secondo in cui invece si trova tutto il data plane con i microservizi. Si può ad esempio avere il cluster con i microservizi e i proxy on premise ed esporre il cluster tramite gateway al control plane gestito dal cloud provider in cloud.



*Immagine 43 - Deployment con external control plane (Istio, n.d.)*

Il deployment tramite multiple control planes è quello più indicato per scenari complessi e distribuiti in termini geografici. Consiste nella composizione di più deployment fra quelli sopra citati. Il fatto di avere molteplici control plane sparsi in differenti cluster consente, in caso di fallimento di control plane o cluster, di non perdere il controllo di tutta l'applicazione ma di solo quei microservizi sui cluster falliti o controllati dai control plane falliti. Non è necessario avere un control plane in ogni cluster, ma un deployment di questo tipo è il più resistente che si può ottenere. I microservizi attraverso i sidecar proxy possono comunque comunicare a cluster diversi tramite l'impiego degli Istio Gateway.



*Immagine 44 - Deployment con multiple control planes (Istio, n.d.)*

In conclusione, ServiceMesh rappresenta un approccio estremamente interessante e piuttosto maturo, la necessità di creare un livello gestione a livello infrastrutturale è evidente in applicativi con molti microservizi. In Istio in particolare è evidente la tendenza a riutilizzare componenti ben

sviluppate all'interno di altri progetti (i proxy Envoy), soluzione che per strumenti così grandi e impattanti sulla disponibilità dei servizi, è fondamentale.

## 4.5 EventMesh

La programmazione e le architetture ad eventi sono concetti fondamentali nello sviluppo software moderno, questo approccio si basa sul concetto di eventi come unità di comunicazione tra i componenti del sistema.

La programmazione ad eventi comporta un modello di programmazione in cui le logiche di business vengono innescate a seguito della ricezione degli eventi invece che essere eseguite in modo sequenziale. Invece che programmare un'elaborazione continua e sincrona si scrive codice per rispondere a determinati eventi che si verificano nel contesto dell'applicazione. Questo permette di avere componenti particolarmente disaccoppiate, con interazioni asincrone, garantendo reattività del sistema e basse latenze nell'elaborazione degli input. Gli eventi inoltre possono essere ascoltati da una molteplicità di riceventi, non rendendo quindi necessario che il produttore di un evento sia in contatto diretto con il consumatore.

Per quanto riguarda le architetture ad eventi il discorso è simile. Nel dettaglio possiamo dire che non sono le chiamate dirette a servizi a scatenare un'azione, quanto la ricezione di un evento da parte di un servizio. Quest'ultimo provvederà poi ad effettuare un'azione nel caso sia stato istruito per gestire quel tipo di evento. In questo scenario è presente un componente chiamato produttore che genera un evento, e un secondo componente chiamato consumatore che ascolta quell'evento. Questo approccio permette di realizzare cooperazioni molto scalabili ed estremamente flessibili anche considerando quanto possono essere disaccoppiati i servizi. Non è infatti necessario avere una API con una firma ben definita per scatenare un'azione, ma sarà solo necessario generare un evento abbastanza descrittivo e completo. Il modo in cui questo evento verrà interpretato non è responsabilità del generatore dell'evento ma del consumatore, consumatori diversi possono ascoltare lo stesso messaggio ma sfrutarne il contenuto in modo diverso.

Questa flessibilità permette di prendersi delle libertà che se non ben calibrate possono dare vita ad uno scenario caotico in cui potrebbero essere generati moltissimi eventi superflui o non ascoltati. Bisogna inoltre prestare particolare attenzione alla distribuzione degli eventi, in questo tipo di

architettura, infatti, è solitamente un event broker che li consegna e non il produttore. Un event broker deve essere in grado di assicurarsi che gli eventi vengano consumati, fornendo eventualmente conferme ai componenti che li hanno generati .

Event mesh è un'infrastruttura dinamica che permette di inviare notifiche (eventi) alle applicazioni in un ambiente distribuito.

L'impiego di un approccio asincrono permette di effettuare richieste a servizi esterni e proseguire l'elaborazione senza attenderne la risposta. Questo approccio garantisce diversi vantaggi come: aumento della scalabilità delle applicazioni, disaccoppiamento fra i servizi e resistenza in caso di fallimento del servizio chiamato. Di contro, un approccio asincrono può aumentare la complessità delle interazioni non trattandosi più di interazioni lineari, può causare problemi di inconsistenza fra sistemi e aumentare la difficoltà nello sviluppo e nel mantenimento del codice.

Nell'ambito dell'architettura guidata dagli eventi (EDA), con evento si intende un cambiamento, un'azione o una rilevazione all'interno di un sistema che genera una notifica. Questa notifica viene poi inviata agli altri sistemi così che possano rispondere all'evento. L'event mesh crea il collegamento tra i sistemi, ma a differenza della ServiceMesh, utilizza gli eventi come mezzo di comunicazione e invece che consegnare direttamente gli eventi al servizio, ne delega la consegna ad un event broker.

La sfida consiste nel capire come spostare, in modo efficiente, scalabile ed economico, i dati in un'infrastruttura che non solo si trova in aree geografiche distanti ma che esiste in cluster separati ed eterogenei. La soluzione è l'event mesh, un'infrastruttura progettata per il trasporto di eventi in qualsiasi ambiente, anche tra più cloud (RedHat, Agosto).

I principali vantaggi di EventMesh sono:

- Scambio di eventi affidabile:

Si tratta di una infrastruttura resistente, offrendo affidabilità nella consegna dei messaggi anche in caso di errore dei consumatori.

- Disaccoppiamento fra servizi:

Lavorando con eventi e non chiamate ad API, ad esempio, è possibile generare un evento completo senza sapere come verrà usato dai consumatori.

- Scalabilità:

L’impiego di intermediari molto sofisticati per la gestione dei messaggi garantisce una capacità di scalare facilmente.

- Osservabilità:

Per ragioni legate all’impiego di gateway simili ai sidecar proxy in ServiceMesh risulta facile effettuare log e monitoraggio di ogni microservizio.

- Sicurezza:

Grazie al supporto dei proxy si possono cifrare le comunicazioni senza aggiungere logica in ogni microservizio

- Performance elevate:

Il modo in cui vengono gestiti i messaggi permette di ottenere delle latenze estremamente basse. Al contempo è possibile distribuire gli eventi in modalità multicast e si ottengono throughput molto più elevati di semplici chiamate ad API.

- Indipendenza dalle tecnologie:

Lavorando con eventi composti da stringhe di testo e comunicando via rete non sono rilevanti le tecnologie, i linguaggi o i runtime usati dai microservizi

EventMesh, pur essendo un progetto diverso da ServiceMesh, agisce con pattern e approcci simili, questo porta con sé molti dei problemi presentati da ServiceMesh, soprattutto legati alla proliferazione dei componenti e la proliferazione delle interconnessioni fra di essi.

EventMesh può non essere l’unico approccio utilizzato per gestire le comunicazioni. È anche possibile affiancare ad EventMesh altri sistemi per il supporto alle comunicazioni fra microservizi come ServiceMesh per quanto riguarda le comunicazioni message driven e via API, avendo così un sistema che sfrutta EventMesh per le comunicazioni asincrone e ServiceMesh per quelle sincrone.

## 4.6 Componenti EventMesh

Le componenti che costituiscono una EventMesh sono particolarmente differenti e non sono costanti in tutte le implementazioni, questo capitolo vuole elencare le principali senza specificare le tecnologie attraverso le quali vengono implementate. Nei capitoli successivi ci concentreremo su un’unica implementazione scendendo quindi nel dettaglio.

#### **4.6.1 Sidecar Proxy o EventMesh Nodes**

Proprio come nel caso della ServiceMesh, anche EventMesh nasconde il microservizio al resto dell’infrastruttura e per includere la logica legata alla comunicazione solo dentro a un livello intermedio e non dentro al microservizio contenente la logica di business. Questo livello intermedio è talvolta implementato attraverso sidecar proxy come nel caso di Solace (una delle principali implementazioni di EventMesh) (Menning, 2022). Altre volte invece si usano EventMesh nodes come nel caso della Apache EventMesh (WeBank, 2023).

A differenza della ServiceMesh, in EventMesh i proxy (o i nodes) non devono occuparsi del costoso compito di gestire il routing e le sue regole, almeno non del tutto, dal momento che EventMesh è direttamente connesso all’event broker, il quale si occupa di trasportare gli eventi siccome nel modello ad EventMesh non sono i proxy ad occuparsi del trasporto degli eventi ma gli event broker.

I sidecar proxy o gli EventMesh nodes sono però anche qui responsabili di stabilire canali cifrati tramite mTLS ed esporre metriche utili per effettuare un monitoraggio accurato.

In EventMesh i proxy però acquisiscono un compito non presente prima, ovvero quello di trasformare una comunicazione non event-driven in event-driven. Non è infatti detto che il microservizio dietro al proxy lavori ad eventi, è quindi necessario creare un significativo livello di conversione per continuare a far lavorare il microservizio a messaggi senza che si accorga della natura ad eventi del resto dell’infrastruttura. Questo è un aspetto fondamentale e complicato da realizzare, ma rappresenta anche un potente strumento dal momento che le architetture e le infrastrutture ad eventi sono ben più capaci di lavorare con basse latenze e con alti throughput rispetto alle architetture tradizionali.

Essendo collegati direttamente all’event broker, sono i proxy a doversi sottoscrivere a determinati tipi di eventi o a dover pubblicare eventi, è quindi importante che siano in grado di interfacciarsi con diversi tipi di event broker.

Le funzionalità dei proxy cambiano in base al tipo di implementazione di EventMesh che scegliamo, non sono però usati proxy esterni come nel caso degli Envoy proxy usati nella Istio ServiceMesh.

Inoltre, come già menzionato, non in tutte le implementazioni sono usati i proxy, talvolta vengono solo offerte librerie con connettori fra linguaggi di programmazione e nodi EventMesh.

#### 4.6.2 Event Broker

Gli event broker sono componenti chiave all'interno di EventMesh. Sono responsabili di gestire e recapitare gli eventi fra microservizi producer e consumer all'interno di uno o più cluster. Gli event broker non fungono solo da intermediari per i messaggi o da instradatori ma si occupano di memorizzare gli eventi e mantenere delle code di eventi da far consumare a chi vi si sottoscrive. Nel dettaglio, il consumo di un evento avviene tramite una notifica a tutti i consumatori che sono sottoscritti a un determinato topic, che leggono ed eseguono una funzione di callback tutte le volte che arriva un nuovo evento con un topic a cui sono sottoscritti.

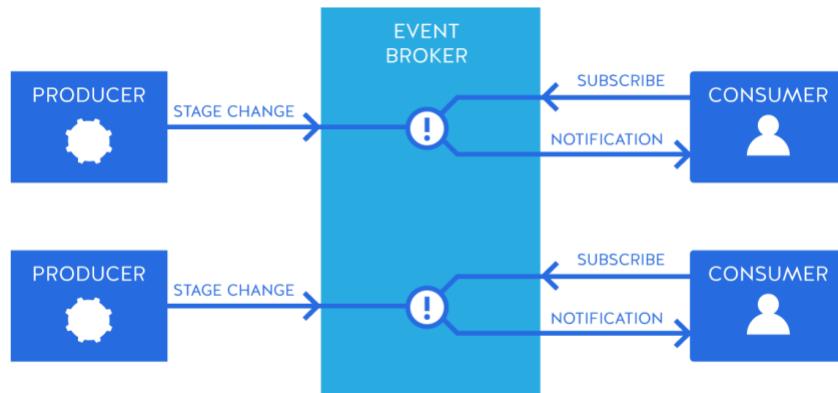


Immagine 45 - Logica di base event broker

Vediamo i compiti principali di un event broker all'interno di una event mesh:

- Smistamento degli eventi:

Dopo aver ricevuto un evento appartenente a un determinato topic l'event broker deve notificare tutti gli utenti sottoscritti a quel topic e inoltrare loro l'evento

- Gestione delle consegne:

Si occupa di eventuali ritrasmissioni e mantiene un evento fintanto che non è stato consegnato a tutti i clienti in ascolto (o aderisce a differenti politiche in merito alla ritenzione degli eventi).

- Scalabilità:

Essendo la spina dorsale delle comunicazioni deve essere facilmente scalabile orizzontalmente mantenendo dati consistenti

- Resistenza ai guasti:

Gli eventi non vanno persi ed è opportuno implementare tecniche che permettano di non perdere gli eventi anche quando istanze di uno stesso broker falliscono (di solito si usano sempre più istanze, spesso distribuite in cluster separati)

- Estensione:

Anche in uno scenario multicluster uno stesso event broker deve essere in grado di gestire la distribuzione degli eventi in modo corretto ed efficiente

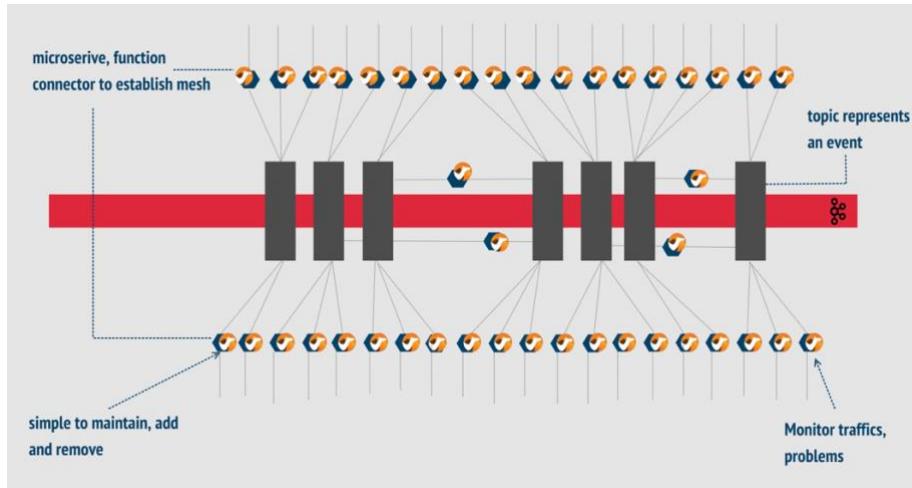
Alcuni esempi di event broker, o MOM (Message Oriented Middleware), sono Apache Kafka, RabbitMQ, Apache Pulsar, Google Cloud Pub/Sub, Apache RocketMQ.

#### **4.6.3 Strumenti di log**

I proxy impiegati in EventMesh, indipendentemente dal fatto che siano realizzati con EventMesh nodes o con sidecar proxy, sono ottimi nodi per raccogliere informazioni sul microservizi con i quali collaborano. La logica è la stessa già vista in ServiceMesh dove i proxy collegati a sistemi di log riportano ad ogni chiamata statistiche sulla chiamata. Anche in EventMesh lo strumento principalmente usato è Prometheus.

La presenza del MOM rappresenta un altro punto di monitoraggio importante, attraverso il MOM possiamo verificare il traffico complessivo della rete e analizzare quali sono i topic più utilizzati o quelli che hanno problemi o mostrano anomalie.

Quanto detto riguarda le caratteristiche principali presenti in quasi tutti i proxy delle diverse implementazioni di EventMesh, implementazioni diverse sfruttano i proxy secondo modi e approcci non sempre uguali.



*Immagine 46 - Generico modello architettonale semplificato EventMesh*

## 4.7 Implementazioni EventMesh

Le implementazioni di EventMesh disponibili oggi sono poche e piuttosto differenti fra loro soprattutto in termini di architettura e componenti impiegate, le principali sono Apache EventMesh, Sap EventMesh, Solace PubSub+ Platform e Knative EventMesh.

Molti dei progetti di implementazione sono ancora in piena fase di sviluppo quindi non tutte le funzionalità presentate potrebbero essere già implementate.

Apache Event Mesh è la principale implementazione open source di EventMesh, verrà trattata in modo approfondito nel prossimo capitolo. SAP Event Mesh, chiamata anche Event-broker-as-a-service da SAP, è un'implementazione minimale di EventMesh, la quale non si discosta molto da un event broker con qualche potenzialità in più. Il servizio in cloud è stato disattivato nel febbraio di quest'anno, al momento sta venendo integrato come funzionalità in un altro prodotto (Strothmann, 2023).

Solace PubSub+ Platform è uno dei progetti di EventMesh apparentemente più maturi, sebbene la documentazione sia presente e ben fatta, non sono disponibili molti dettagli in merito all'implementazione. Knative EventMesh invece non realizza esattamente una EventMesh ma piuttosto fornisce componenti e API per impiegare il Knative Broker in maniera da costituire un EventMesh molto semplice.

## 4.8 Apache EventMesh

AEM (Apache Event Mesh) è la principale implementazione open source di EventMesh. Apache EventMesh è nato a WeBank, una banca fondata nel 2014 da Tencent, un conglomerato industriale particolarmente attivo nel campo della tecnologia e dell'intrattenimento. Oltre a WeBank i principali collaboratori in fase di sviluppo sono stati Oppo e Huawei. Nel 2019 è diventato un progetto open source su GitHub ed oltre ad essere da tempo parte dell'Apache Incubator, nella primavera di quest'anno (2023) è diventato un Top Level Project dell'Apache Foundation. Il progetto è ancora in fase di sviluppo, molte delle funzionalità presentate non sono state implementate e alcune di quelle implementate non sono completamente compatibili con le componenti a cui si legano. Secondo la documentazione però è possibile creare un'EventMesh con AEM sia in un ambiente locale che in cloud.

AEM si è concentrato nel creare una implementazione di EventMesh decentralizzata orientata a deployment multi-cluster. Le tecnologie impiegate sono spesso legate ad altri strumenti sviluppati da AliBaba, come ad esempio l'event broker che utilizza: RocketMQ.

Le principali funzionalità di AEM sono:

- Utilizzo CloudEvents:

AEM impiega come specifica per gli eventi i CloudEvents, una specifica della CNCF, che descrivere le regole per codificare un evento e librerie per la loro interpretazione in svariati linguaggi.

- Connessioni con molti MOM:

AEM offre connessioni con molteplici MOM, rendendo così possibile scegliere quello più adatto in base ai casi d'uso, il connettore più affidabile resta quello per RocketMQ.

- Deployment multi-cluster:

Permette di distribuire eventi fra deployment differenti e localizzati in molteplici cluster.

### 4.8.1 EventMesh Runtime

EventMesh Runtime è il componente fondamentale di AEM, spesso viene anche chiamato EventMesh Node.

Si occupa di interfacciare ogni microservizio (o gruppi di microservizi) con l'event broker. A differenza del sidecar proxy di ServiceMesh è più semplice dal momento che non gestisce politiche di routing. A differenza dei proxy in ServiceMesh però può essere necessario che si occupi di tradurre una comunicazione a messaggi sincroni con il servizio mentre dall'altro lato mantiene un rapporto asincrono a eventi con l'event broker.

Nel caso di AEM la funzionalità principale, ovvero la funzione di traduzione della comunicazione, deve essere manualmente implementata dall'utente dal momento che è fortemente legata alla natura e alla tipologia del servizio.

Le modalità di interazione fra Runtime e Client possono essere di tre tipi:

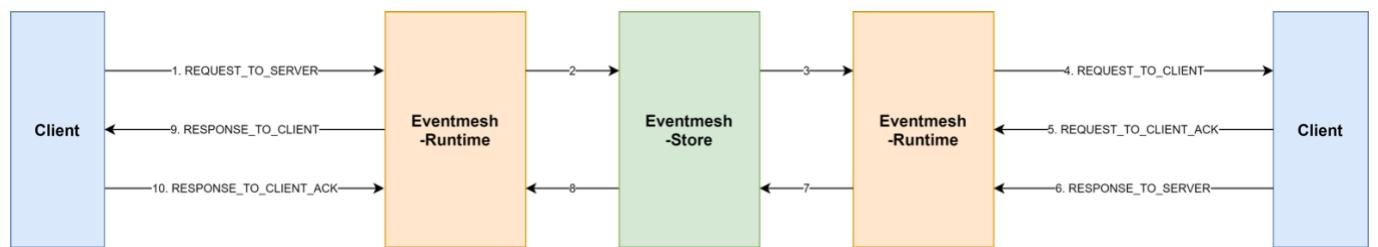


Immagine 47 - Messaggi sincroni (Apache EventMesh, n.d.)

I messaggi sincroni prevedono risposta, ed è quindi necessaria un'interazione speculare in cui tutte le azioni svolte da un client vengono svolte anche dal suo pari ma successivamente al primo messaggio e con il fine di fornire una risposta. In questo tipo di interazione vengono forniti anche ACK dai client agli EM Runtime (o EM Nodes).

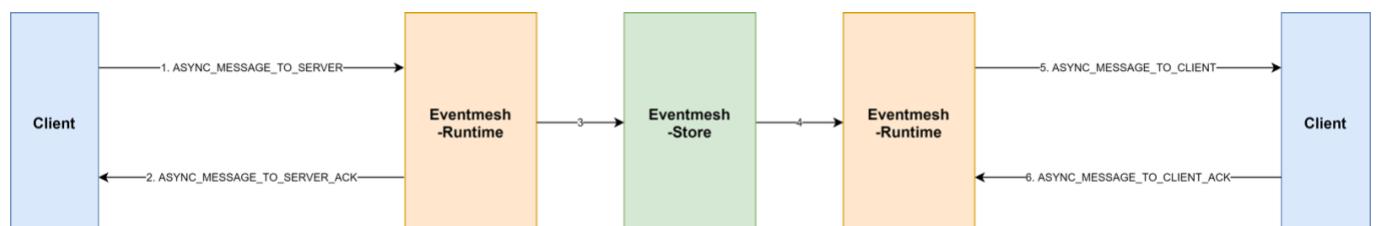


Immagine 48 - Messaggi asincroni (Apache EventMesh, n.d.)

Nei messaggi asincroni, non essendoci necessità di ottenere alcuna risposta si ha una comunicazione molto più snella e le interazioni sono dimezzate dal momento che ad ogni messaggi non deve corrispondere una rispettiva risposta. Le conferme di ricezione in questo caso sono generate dagli EM Runtime e destinate ai client.

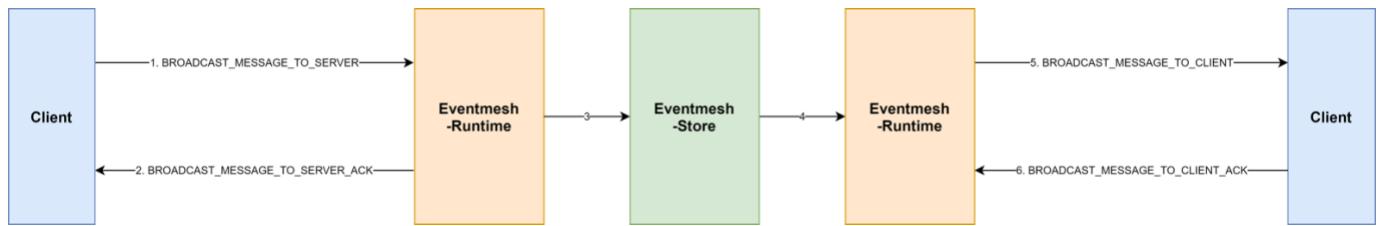


Immagine 49 - Messaggi broadcast (Apache EventMesh, n.d.)

I messaggi broadcast si distinguono a quelli asincroni solo per il compito svolto dall'event broker. Nel caso di messaggi broadcast, infatti, l'event broker invia a tutti gli EM Runtime lo stesso messaggio, e non solo a uno come nel caso dei messaggi punto a punto. Anche in questo tipo di interazione le conferme di ricezione sono generate dagli EM Runtime e destinate ai client.

I protocolli di comunicazione supportati fra microservizio ed EM Runtime sono 3: TCP, HTTP, gRPC.

Per quanto riguarda invece il lato client sono presenti SDK in vari linguaggi fra cui: Java, C/C++, Go, Python, Rust ed altri. Al momento l'unico implementato è Java.

#### 4.8.2 EventMesh Store

L'event store in AEM è a tutti gli effetti un MOM al quale si collegano gli EM Runtime. L'installazione segue le procedure del MOM scelto e può essere visto come componente esterno a EM al quale EM si appoggia. Nel caso di EventMesh il MOM scelto è stato RocketMQ.

In generale i MOM hanno caratteristiche simili, tra cui latenze basse, ampio throughput, sono molto scalabili e tolleranti ai guasti.

Lo storage layer (event store) può essere implementato da RocketMQ, Apache Kafka, Apache Pulsar, RabbitMQ, Redis, MySQL, Pravega o Apache IoTDB. Di tutti questi solo i connettori usati per collegare gli EM Nodes e RocketMQ sono presenti.

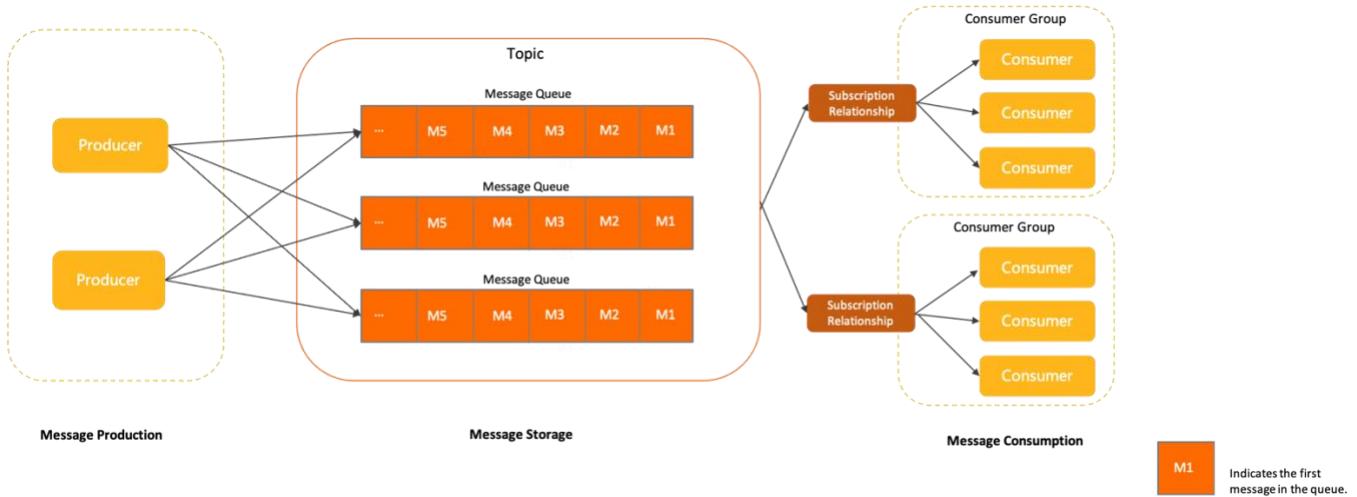


Immagine 50 - Architettura dell'Event Store (RocketMQ)

I messaggi vengono raggruppati per topic, qualsiasi producer può produrre qualsiasi tipo di topic e qualsiasi consumer può leggere qualsiasi topic (a patto che sia sottoscritto a quel topic).

#### 4.8.4 EventMesh Schema Registry

Si tratta di un server che offre RESTful API come interfaccia per raccogliere e condividere Schemas inviati dai client. Gli Schemas sono le interfacce serializzate degli eventi che possono essere generati dai client. Oltre a memorizzare le interfacce degli eventi può anche effettuare validazione di eventi o processi di serializzazione e de serializzazione.

La base dello schema registry è OpenSchema, che fornisce la maggior parte delle specifiche relative alle API e alle funzionalità.

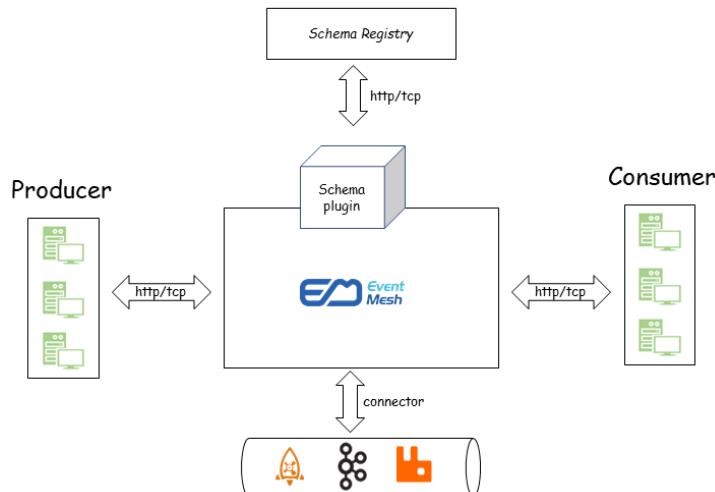


Immagine 51 - Architettura Schema Registry (Apache EventMesh, n.d.)

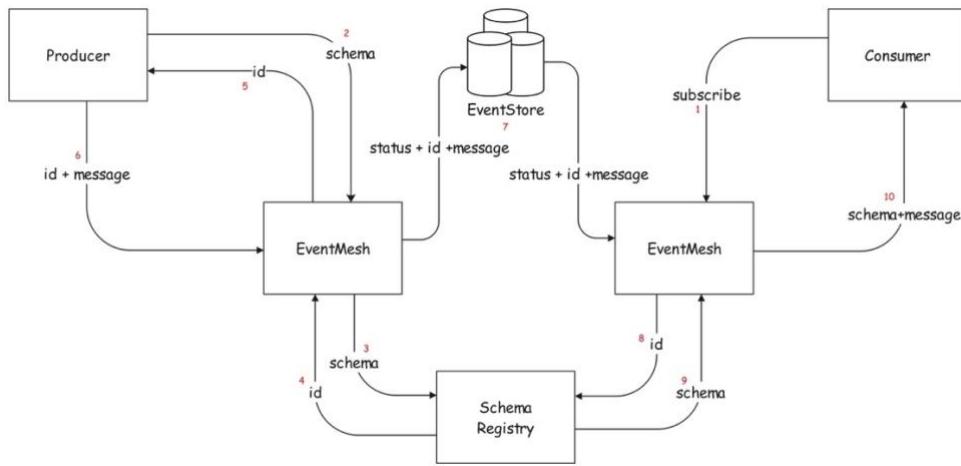


Immagine 52 - Modello di interazione con Schema Registry (Apache EventMesh, n.d.)

Lo schema registry non è ancora stato sviluppato e alcune proposte in merito al design stanno ancora venendo considerate.

#### 4.8.5 EventMesh Metrics

L'aspetto delle metriche è fondamentale ed è uno degli aspetti maggiormente apprezzati di EventMesh.

Per quanto riguarda il monitoraggio AEM si appoggia a OpenTelemetry, ovvero una collezione di strumenti, API e SDK volta a generare, raccogliere ed esportare log e metriche.

Il secondo tassello fondamentale per il monitoraggio di AEM è Prometheus, soluzione già adottata da Istio ServiceMesh, offre plugin che permettono l'interfacciamento con OpenTelemetry e tramite Grafana offre uno strumento completo dalla raccolta fino alla visualizzazione e all'analisi dei dati.

Al momento, nulla in merito alle metriche è stato implementato e gli unici due requisiti presenti definiscono la necessità di osservare traffico HTTP e TCP tramite Prometheus.

In conclusione, è importante ricordare come EventMesh non sia da intendere come unico approccio alla comunicazione, ma sempre in coppia a sistemi di comunicazione non event driven come ad esempio ServiceMesh. In scenari in cui la semantica delle operazioni è facilmente implementabile tramite eventi è sensato e proficuo l'impiego di questo tipo di tecnologia. Dettagli sull'effettivo utilizzo verranno spiegati nel capitolo successivo.

## 5 Progetto e implementazione di un Event Mesh

L'obiettivo dell'implementazione è stato creare una proof of concept di EventMesh, sulla quale effettuare benchmark per capire qualora questa tecnologia possa sopportare carichi di lavoro considerevoli e in che modo.

L'architettura su cui è stato sviluppato il prototipo è un'architettura basata su 4 host fisici su cui è stato installato un cluster K3S.

— Connessione LAN  
— Connessione iLO

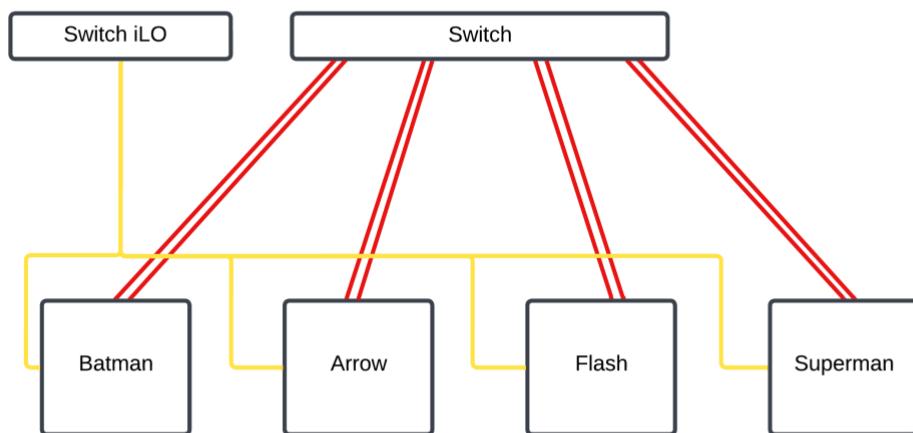


Immagine 53 - Architettura di rete per test

Questa parte di infrastruttura è fisicamente ospitato all'interno del laboratorio di Ricerca e Sviluppo di ImolaInformatica.

Host	OS	RAM	CPU	Ethernet	Server	Storage
Batman	Ubuntu 22.04.2 LTS	11917MiB	Intel Celeron G1610T (2) @ 2.300GHz	2x1Gb/s + 100Mb/s iLO connection	ProLiant MicroServer Gen8	500GB HDD
Arrow	Ubuntu 22.04.2 LTS	11917MiB	Intel Celeron G1610T (2) @ 2.300GHz	2x1Gb/s + 100Mb/s iLO connection	ProLiant MicroServer Gen8	500GB HDD

Flash	Ubuntu 22.04.2 LTS	11917MiB	Intel Celeron G1610T (2) @ 2.300GHz	2x1Gb/s + 100Mb/s iLO connection	ProLiant MicroServer Gen8	500GB HDD
Superman	Ubuntu 22.04.2 LTS	11917MiB	Intel Celeron G1610T (2) @ 2.300GHz	2x1Gb/s + 100Mb/s iLO connection	ProLiant MicroServer Gen8	500GB HDD

*Immagine 54 - Tabella specifiche host*

Tutto il codice e gli esiti dei benchmark sono disponibili su GitHub a <https://github.com/Leodom01/EventMeshThesis>.

## 5.1 Cluster Kubernetes

È stato impiegato un cluster K3S, una distribuzione di cluster Kubernetes sviluppata con in mente il mondo IoT e il cloud computing.

Le caratteristiche di K3S sono quelle di essere una versione molto leggera di Kubernetes, con poche richieste in termini di memoria, storage e capacità computazionali. Tutto il software è contenuto in un file binario di 60MB e il deployment iniziale impiega circa 30 secondi per essere eseguito. Inoltre, include CRI, CNI, load balancer e altre componenti spesso usate ma non disponibili nelle installazioni stock di Kubernetes.

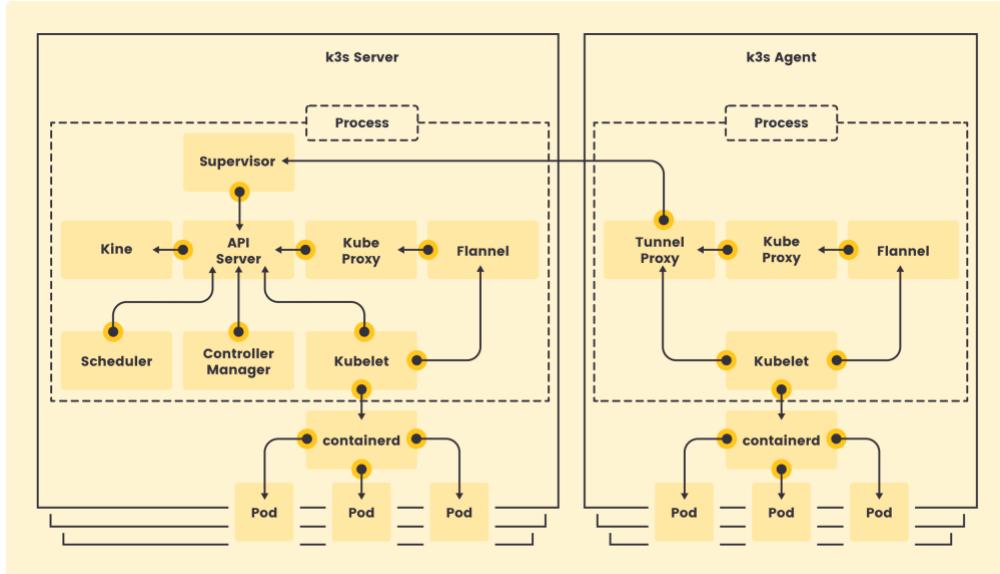


Immagine 55 - Architettura agent e manager K3S (How it Works , n.d.)

L’architettura usata è composta da tutti e quattro gli host elencati sopra, Batman è l’unico nodo master e ospita il control-plane, tutti gli altri sono nodi worker.

## 5.2 Architettura

Sebbene i concetti restino gli stessi di Apache EventMesh e delle altre implementazioni di EventMesh, i linguaggi di programmazione impiegati e alcuni dettagli implementativi sono cambiati rispetto all’implementazione di Apache. A livello architettonico non ci sono molte differenze, la struttura è composta da un pod per ogni microservizio e un MOM esterno al cluster Kubernetes. Ogni pod oltre al servizio contenente la logica dell’applicazione, contiene anche un sidecar proxy.

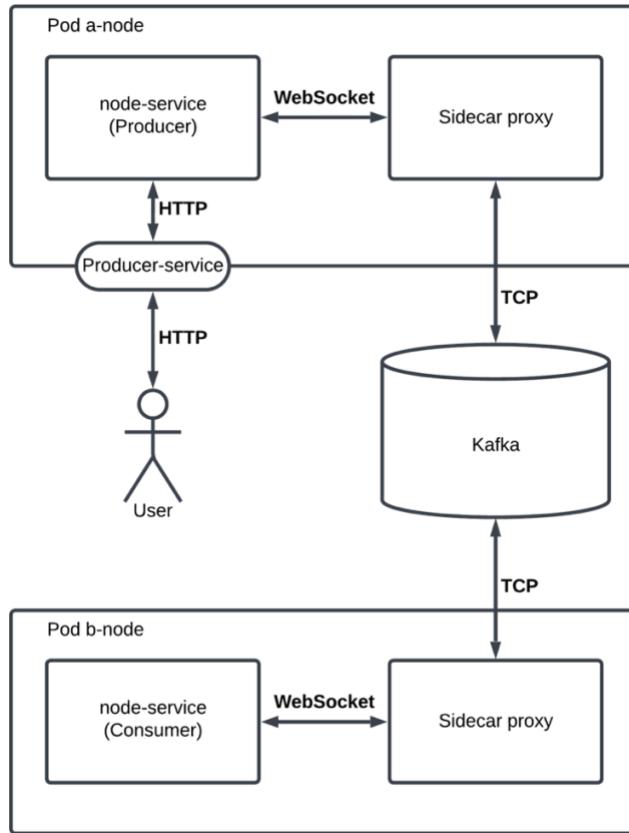


Immagine 56 - Architettura della mia implementazione di EventMesh

### 5.2.1 Producer

Il producer è realizzato tramite server NodeJS, dispone di una API /send esposta tramite Express ed è raggiungibile tramite un Service Kubernetes.

L'API /send si occupa di generare una richiesta HTTP di questo tipo:

```
var httpRequest = new http.IncomingMessage({
  method: 'GET',
  url: target,
  headers: {
    'Content-Type': 'application/json',
    'Origin': serviceName,
    'X-Request-ID': requestID
  },
  rawHeaders: ['Content-Type', 'application/json']
});
httpRequest.data = data
```

Immagine 57 - Richiesta HTTP del Producer

La richiesta è stata fatta per simulare una connessione con il Consumer via HTTP, in modo da rendere necessario per il proxy tradurre la comunicazione in una comunicazione ad eventi prima di inoltrare l'evento al broker.

Nel dettaglio le variabili usate sono:

- target:

Si tratta dell'URL del server Consumer, target può essere acquisito tramite il parametro “destination” della API /send, un esempio di target è `http://conusmerSercvice/myPath`.

- serviceName:

Il nome del servizio preso dalla variabile d'ambiente `SERVICE_NAME` impostata direttamente dalle specifiche (campo `specs`) del container nel file YAML di configurazione del pod. In questo modo è facile creare tanti servizi producer differenti cambiando solo una stringa nel file di configurazione su Kubernetes. Se la variabile non è impostata viene usata la stringa “`undefined`”.

- requestID:

È un UUIDv4 generato dal package `uuid` di Node, permette di identificare in modo univoco una richiesta HTTP per gestire ACK e monitoraggio.

- data:

È il body del messaggio, al momento è impostato un placeholder non rilevante.

La comunicazione fra producer e sidecar proxy avviene in HTTP attraverso WebSocket, si tratta di un protocollo facilmente implementabile e che offre ad entrambi gli interlocutori la possibilità di prendere iniziativa nell'invio dei messaggi, così facendo la comunicazione risulta dinamica, efficiente e facile da utilizzare.

Dopo averlo inviato al proxy, il messaggio viene salvato in una mappa che usa come chiave gli UUID delle richieste e come valore il messaggio stesso. Non appena verrà ricevuto l'ACK da parte del proxy allora l'UUID presente nell'ACK verrà usato per rimuovere la relativa richiesta dalla tabella delle richieste in corso. Questa tabella può poi essere impiegata per effettuare tentativi di riconnessione e per avere sempre sotto controllo il numero di messaggi inviati ma non ancora confermati.

Il producer condivide i messaggi facendo passare tutto il traffico rilevante attraverso il sidecar proxy con WebSocket.

Un altro approccio più strutturale consiste nel ridirezionare tutto il traffico supportato dal proxy al sidecar proxy lavorando totalmente all'insaputa del servizio, seguendo l'approccio usato in Istio ServiceMesh.

### **5.2.2 Consumer**

Anche il consumer è realizzato in NodeJS, sebbene il suo comportamento sia quasi del tutto analogo a quello del producer alcuni aspetti sono differenti. In particolare, le operazioni effettuate dopo la ricezione di un messaggio via WebSocket sono differenti, questo è dovuto al fatto che producer e consumer implementano logiche di business diverse.

Le informazioni che possono essere ricevute tramite la WebSocket possono essere ACK o messaggi.

Nel caso dell'ACK il comportamento è il medesimo del producer.

Nel caso invece si tratti di un messaggio vero e proprio il consumer può liberamente decidere come implementare la risposta. Essendo questo consumer sviluppato per effettuare benchmark generando risposte ai messaggi ping del producer, il codice si occupa semplicemente di cambiare mittente e destinatario nella richiesta HTTP ricevuta (quella generata dal producer) e modificare l'UUID aggiungendo in testa la lettera R, rendendo più immediata l'identificazione del traffico di ritorno in fase di analisi dei dati.

### **5.2.3 Proxy**

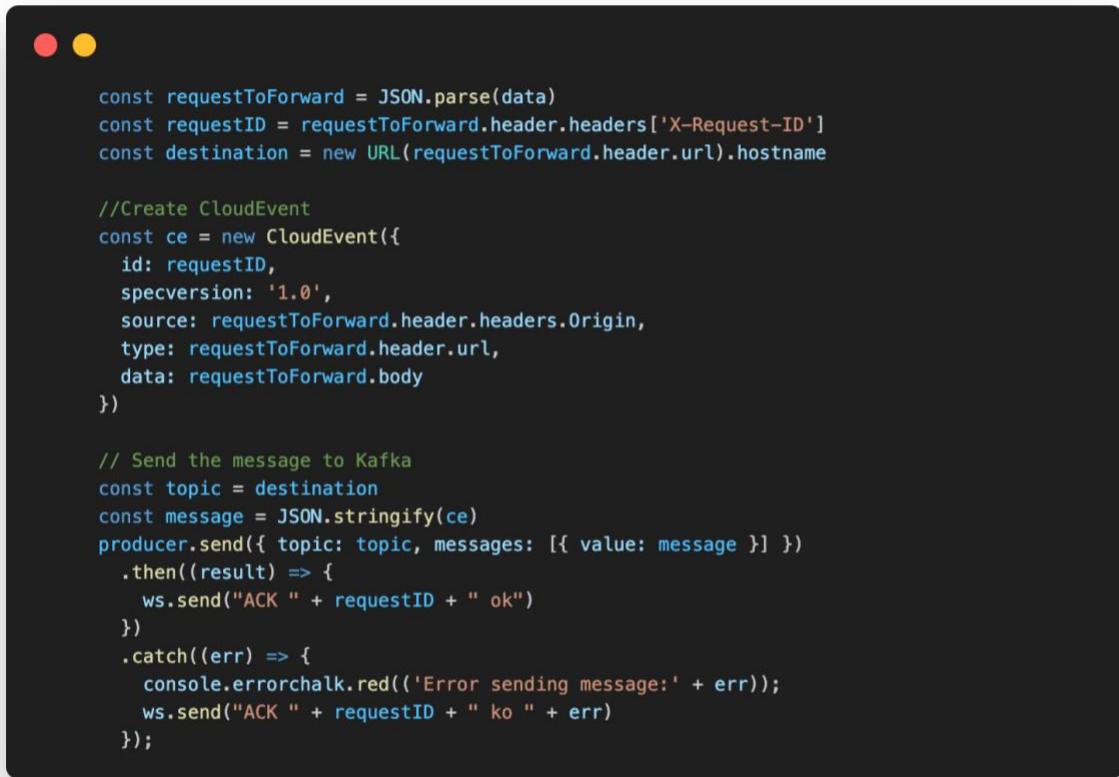
Anche il proxy è realizzato in NodeJS, ma dal momento che tutte le interazioni avvengono tramite rete, né il linguaggio del proxy né quello dei servizi che lo sfruttano sono rilevanti ai fini della compatibilità fra i due.

In questa implementazione si è deciso di avere un sidecar proxy inserito nello stesso pod dell'application container.

Il proxy ha il compito di creare la WebSocket e di collegarsi al MOM, la connessione con l'application container avviene tramite WebSocket, mentre la connessione con Kafka attraverso funzioni della libreria kafkajs di JavaScript.

NodeJS offre librerie per creare client della maggior parte dei MOM: Kafka, RocketMQ, RabbitMQ, Pulsar e molti altri, rendendo quindi l'impiego di MOM alternativi poco complesso.

Il compito principale del proxy in questo caso è quello di tradurre richieste HTTP in eventi CloudEvents da inviare al MOM, che a sua volta li inoltrerà ai proxy dei servizi destinatari.



```
const requestToForward = JSON.parse(data)
const requestID = requestToForward.header.headers['X-Request-ID']
const destination = new URL(requestToForward.header.url).hostname

//Create CloudEvent
const ce = new CloudEvent({
  id: requestID,
  specversion: '1.0',
  source: requestToForward.header.headers.Origin,
  type: requestToForward.header.url,
  data: requestToForward.body
})

// Send the message to Kafka
const topic = destination
const message = JSON.stringify(ce)
producer.send({ topic: topic, messages: [{ value: message }] })
  .then((result) => {
    ws.send("ACK " + requestID + " ok")
  })
  .catch((err) => {
    console.error(chalk.red('Error sending message: ' + err));
    ws.send("ACK " + requestID + " ko " + err)
  });
};
```

Immagine 58 - Parsing del messaggio HTTP e invio a Kafka

Questo codice mostra la procedura che viene usata quando un messaggio in arrivo tramite la WebSocket deve essere convertito in evento ed inoltrato a Kafka.

I passaggi sono 3: parsing del messaggio HTTP, creazione del CloudEvent tramite la libreria cloudevents, invio a Kafka con gestione della risposta e ACK al servizio.

L'ACK viene sempre inviato al servizio, sia in caso il messaggio sia stato correttamente consegnato a Kafka sia nel caso opposto, nel primo l'ACK conterrà la stringa "ok" e in caso di errore conterrà la stringa "ko"

Ecco il significato delle variabili:

- requestID: l'UUID della richiesta generato dal producer.
- destination: il nome del servizio destinazione, ottenuto estrapolando l'hostname dall'url.
- source: nome del servizio, serve al destinatario per conoscere l'interlocutore.
- type: l'URL intero nel caso il messaggio del producer sia indirizzato a un path specifico.
- topic: la politica dell'implementazione è di usare come topic con il quale caricare i messaggi sul MOM il nome del servizio che dovrà riceverlo.

Per quanto riguarda invece la ricezione di eventi da parte di Kafka il nocciolo dell'elaborazione è contenuto nella funzione asincrona che viene invocata ogni qualvolta arrivi un messaggio da Kafka.

Il proxy si sottoscrive al topic di interesse per il servizio, come spiegato prima il topic appartenente a un servizio è semplicemente il nome del servizio stesso. Il nome del servizio viene passato al proxy secondo la stessa modalità usata nel container del servizio, ovvero tramite la variabile d'ambiente SERVICE\_NAME, impostata anche in questo caso nel file di configurazione del pod su Kubernetes.

```
async function eachMessageHandler({ topic, partition, message })
{
    const msgJson = JSON.parse(message.value)

    //Creazione messaggio HTTP
    var httpRequest = new http.IncomingMessage({
        method: 'GET',
        url: msgJson.type,
        headers: {
            'Content-Type': 'application/json',
            'Origin': msgJson.source,
            'X-Request-ID': msgJson.id
        },
        rawHeaders: ['Content-Type', 'application/json']
    });
    httpRequest.data = msgJson.data

    var toSend = {
        header: httpRequest.socket,
        body: httpRequest.data
    }

    //Invio messaggio ricostruito al servizio
    serviceWs.send(JSON.stringify(toSend))
}
```

*Immagine 59 - Funzione chiamata ad ogni ricezione di messaggio da Kafka*

Una volta che l'evento dal MOM arriva al proxy, il proxy lo riconverte in un messaggio HTTP in modo analogo a quello già visto sopra e lo inoltra al consumer via WebSocket.

```
kind: Deployment
metadata:
  name: a-node
spec:
  replicas: 1
  selector:
    matchLabels:
      app: a-node
  template:
    metadata:
      labels:
        app: a-node
    spec:
      containers:
        - name: node-service
          image: batman.imolab.it:5000/node-service
          env:
            - name: SERVICE_NAME
              value: "a_node"
            - name: PROXY_WEBSOCKET_PORT
              value: "80"
            - name: PROXY_ENDPOINT
              value: "localhost"
          ports:
            - containerPort: 8080
        - name: kafka-proxy
          image: batman.imolab.it:5000/kafka-proxy
          env:
            - name: KAFKA_ENDPOINT
              value: "kafka-service.default.svc.cluster.local"
            - name: KAFKA_PORT
              value: "9092"
            - name: SERVICE_NAME
              value: "a_node"
          ports:
            - containerPort: 80
```

Immagine 60 - YAML di deployment di un servizio e il suo proxy

Questo è il deployment standard in cui viene impiegato un servizio generico chiamato “node-service” che espone la porta 8080 per esporre le proprie API mentre il proxy espone solo la porta 80 essendo quella necessaria al server WebSocket.

#### 5.2.4 EventBroker

Come event broker è stato scelto Apache Kafka.

Apache Kafka è un event broker nato nel 2010 a LinkedIn, rilasciato poi nel 2011 con il supporto dell'Apache Foundation è diventato uno dei principali event broker sul mercato.

In particolare, offre un supporto molto ampio in termini di librerie e connettori disponibili per l'integrazione con molteplici linguaggi. L'aspetto dell'integrazione è particolarmente importante perché permette di implementare i proxy in linguaggi diversi e questo può essere comodo per sperimentare differenti linguaggi in fase di sviluppo.

Kafka garantisce inoltre ottime prestazioni in termini di throughput e latenze nell'ordine delle decine o unità di millisecondi, se non sotto i millisecondi in alcuni scenari. Le performance sono fortemente dipendenti dall'architettura, dal cluster e dalle impostazioni scelte.

Per integrare i proxy JavaScript con Kafka è stata utilizzata la libreria KafkaJS.

```
//Kafka connector setup
const kafkaHostname = process.env.KAFKA_ENDPOINT
const kafkaPort = process.env.KAFKA_PORT
const serviceName = process.env.SERVICE_NAME
const kafka = new Kafka({
  clientId: serviceName+'_proxy',
  brokers: [kafkaHostname+':'+kafkaPort]
});

const producer = kafka.producer();
const consumer = kafka.consumer({ groupId: serviceName+'_groupID'
});
async function run(topicToListenTo) {

  //Consumer setup
  await producer.connect();
  console.log(myChalk.yellow('Producer side connected\n'));
  //Consumer setup
  await consumer.connect();
  console.log(myChalk.yellow('Consumer side connected\n'));
  await consumer.subscribe({ topic: topicToListenTo});
  console.log(myChalk.bgGreen("Added topic to monitor/sub: " + topicToListenTo))

  //Avvio consumer con callback per messaggio in ingresso
  await consumer.run({
    eachMessage: eachMessageHandler
  })
}
```

Immagine 61 - Setup e connessione a Kafka dentro al proxy

Dal momento che il proxy si occupa sia di inoltrare che di ricevere messaggi dal broker è necessario impostare sia un consumer che un producer per Kafka.

Anche qui i riferimenti all'endpoint di Kafka e il nome del servizio (con cui creare un consumer group su Kafka) vengono ottenuti tramite variabili d'ambiente, impostabili tramite file YAML.

Dopo aver collegato il producer è anche necessario collegare il consumer, sottoscriversi al topic contenente i messaggi destinati al microservizio e poi impostare la funzione di callback “eachMessageHandler” che viene invocata ogni qualvolta arriverà un messaggio.

### 5.3 Interazione componenti

L’interazione fra le componenti è quella definita da EventMesh, quindi prevedere l’interazione dell’application container con il proxy, il quale inoltra gli eventi al MOM. Successivamente il MOM si occupa di inviare gli eventi destinati a un dato servizio al suo proxy, il quale invia i messaggi all’application container dopo averli trasformati in messaggi del protocollo che sta venendo utilizzato fra i due servizi. La conversione da evento a protocollo usato dai servizi è necessaria siccome i proxy comunicano fra di loro tramite eventi.

La logica di interazione mostrata qui nel dettaglio è la medesima usata durante i test. L’interazione end-to-end vista dal punto vista dei servizi comporta l’invio di un messaggio di ping usando HTTP invocato tramite API /send sul producer. Il messaggio di ping verrà recapitato al consumer che invierà una risposta HTTP, con UUID uguale a quello della richiesta ma con “R” davanti.

Durante la spiegazione verranno usati i termini consumer e producer per indicare il servizio che genera i messaggi di richiesta e quello che genera messaggi di risposta. In questa e nelle altre implementazioni di EventMesh non è necessario avere un producer e un consumer, ogni nodo può ricoprire sia l’uno che l’altro ruolo. Questo è possibile dal momento che il proxy include sia la logica per consumare gli eventi sia quella per produrli.

In una comunicazione reale il messaggio di risposta conterrebbe la risposta effettiva, e non lo stesso messaggio con UUID cambiato. È bene ricordare però che in molti casi, le applicazioni basate su eventi sono asincrone, ciò significa che i servizi chiamanti non attendono le risposte dai servizi chiamati per proseguire con altre operazioni (AWS, s.d.). Questa implementazione di event mesh offre però conferme di consegna per ogni step del processo e anche conferme end-to-end.

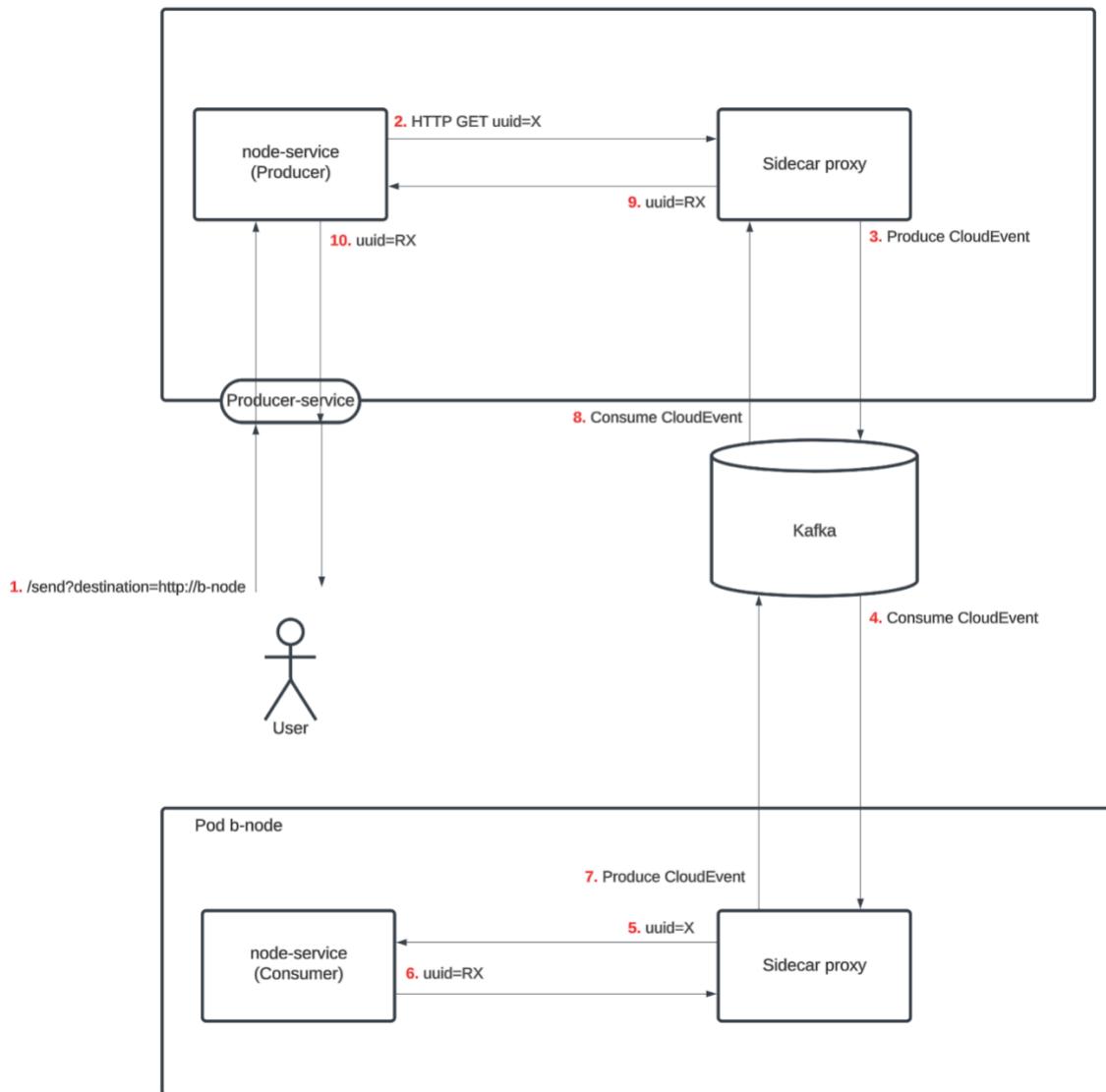


Immagine 62 – Interazioni durante il ping

## 5.4 Benchmark

I benchmark effettuati sull'applicazione si basano sull'effettuare con una certa frequenza richieste di ping attraverso la logica sopra mostrata.

Le metriche registrate permettono di estrarre la durata dei singoli passaggi dell'interazione, nel dettaglio vengono identificati 7 momenti diversi:

- SEND: invocazione dell'API /send sul producer.

- SENT: invio del messaggio da parte del producer sulla WebSocket condivisa con il proxy.
- ACK: messaggio ricevuto e inviato correttamente dal proxy a Kafka.
- GOT: ricezione di un nuovo messaggio (se il messaggio ha UUID che inizia con “R” allora si tratta di risposta a un messaggio di ping).
- TO KAFKA: ingresso di un messaggio diretto a Kafka dentro al proxy.
- FROM KAFKA: ingresso di un messaggio proveniente da Kafka dentro al proxy.
- RES: invio del messaggio di risposta a un ping da parte del consumer.

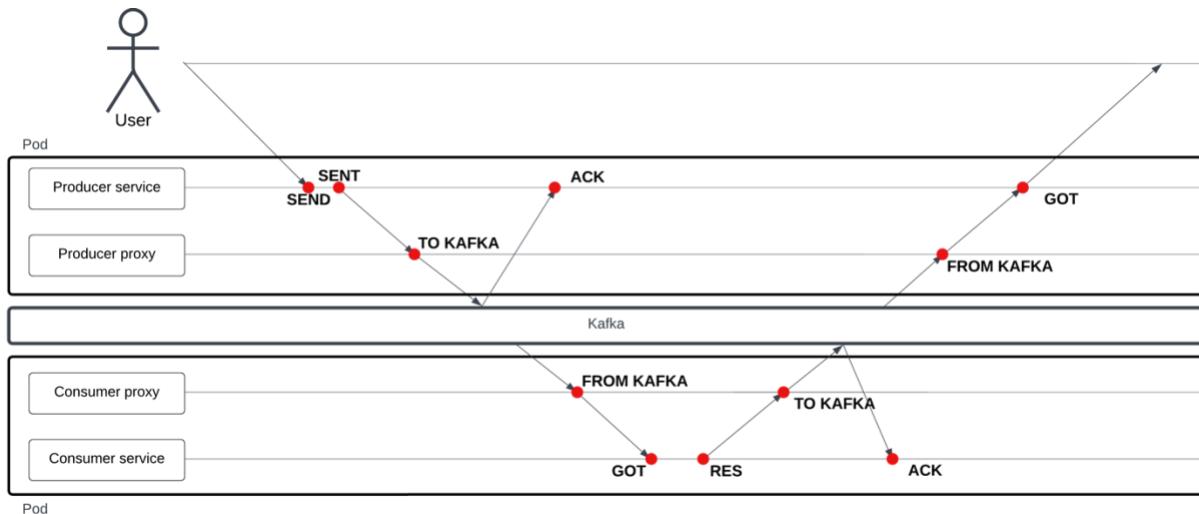


Immagine 63 - Generazione eventi di log

#### 5.4.1 Send Singola

Questo primo test ha come scopo quello di testare il comportamento dell’implementazione in una situazione normale con un carico basso e costante.

La generazione delle richieste è effettuata da una macchina Ubuntu, presente nello stesso cluster di EventMesh, tramite l’impiego di curl. Per evitare problematiche legate ai timestamp dei log causati dalla non perfetta sincronizzazione degli orologi sui vari host del cluster, entrambi i pod sono stati avviati sullo stesso nodo. Dal momento che le chiamate per ottenere la data e l’ora tramite l’oggetto Date di NodeJS ottengono l’ora esatta dal kernel dell’host, tutti i container in entrambi i pod faranno riferimento allo stesso orologio sulla stessa macchina essendo così intrinsecamente consistenti.

Nei i test mostrati anche il deployment Kafka è stato realizzato tramite Kubernetes e il suo pod si trova sullo stesso nodo dei pod consumer e producer.

L'implementazione è stata testata anche su nodi diversi con successo e le uniche differenze in termini di performance sono legate al tempo necessario per instradare e traferire i pacchetti da un nodo all'altro.

Il test Send Singola è stato realizzato effettuando chiamate all'API /send in modo sincrono e bloccante.

Alla luce della natura fortemente asincrona della comunicazione ad eventi si è deciso di restituire come risultato alla chiamata all'API il risultato fornito dall'event broker in merito alla ricezione dell'evento, durante i test questo evento viene loggato tramite la stringa ACK.

Tutti i grafici mostrati usano come unità di misura i millisecondi, un punto (X; Y) rappresenta un messaggio la cui SEND è stata registrata a X ms dall'inizio del test e l'interazione presa in esame è durata Y ms.

La linea nera all'interno dei grafici è una media mobile a 10 punti (utile per ridurre il rumore).

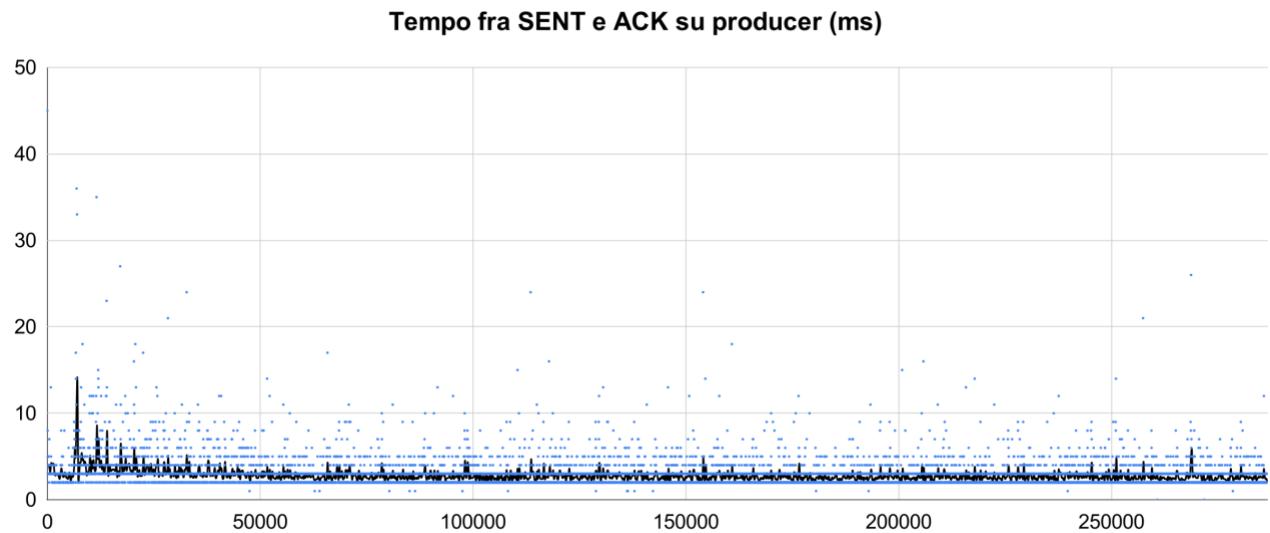
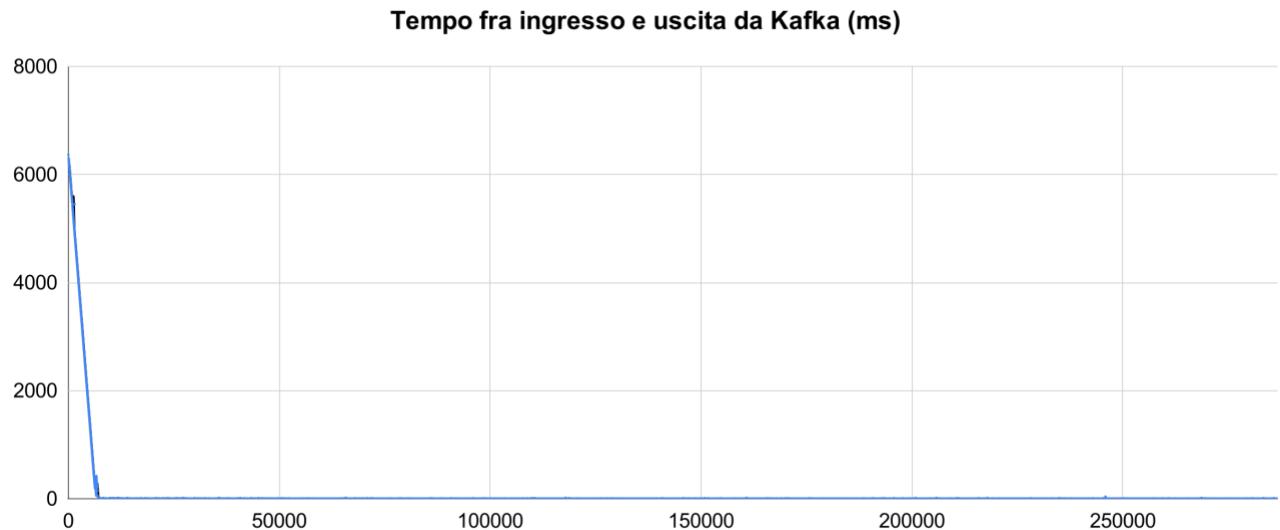
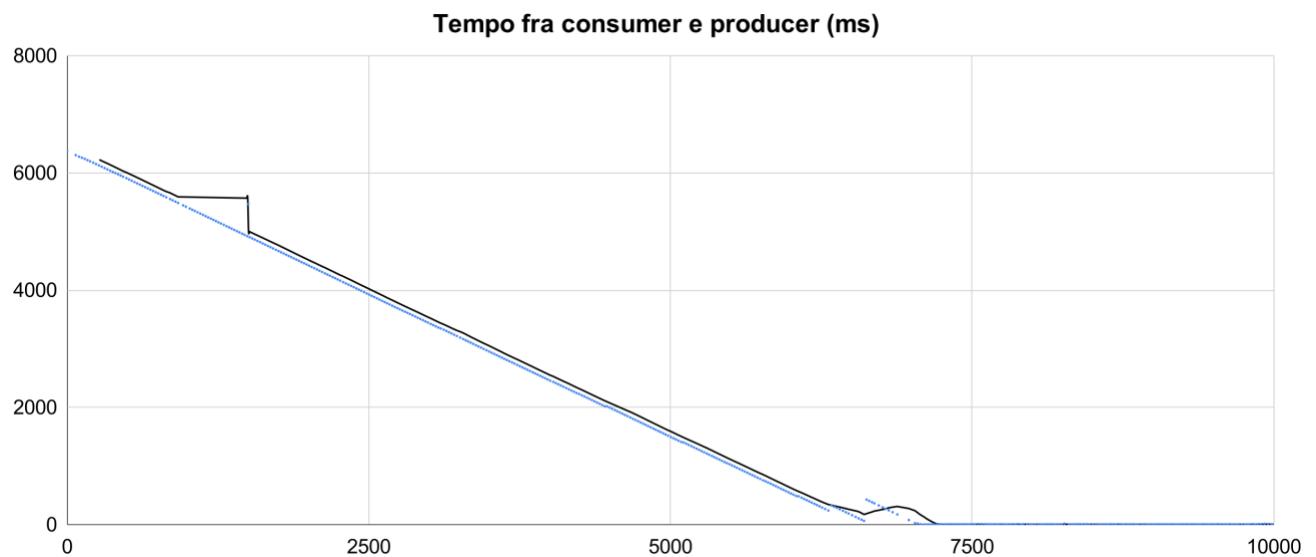


Immagine 64 - Tempo intercorso fra l'invio del messaggio al proxy e la conferma di consegna da parte di Kafka: SENT-ACK



*Immagine 65 - Tempo impiegato da Kafka per inoltrare il messaggio: TO\_KAFKA-FROM\_KAFKA*



*Immagine 66 - Tempo totale per la trasmissione di un messaggio: SEND-GOT (solo i primi 10 secondi)*

Alcuni degli ulteriori dati registrati sono i seguenti (statistiche elaborate su tutte le oltre 13000 entry di log):

- SEND-SENT: tempo necessario alla creazione del messaggio HTTP e al suo invio.  
Media: 0.27ms Varianza: 0.21
- FROM\_KAFKA-GOT: tempo intercorso fra l'ingresso di un nuovo evento nel proxy e la ricezione del corrispondente messaggio dal consumer.

Media: 1.55ms Varianza: 11.96

Analizzando i tre grafici possiamo notare come sia significativo l'impatto di Kafka sulla latenza della comunicazione.

Da circa 7.5 secondi dopo il primo messaggio, la comunicazione si stabilizza su una latenza sotto i 5ms, compatibile con le latenze minime raggiungibili in Kafka.

Si tratta di un risultato estremamente positivo e interessante, resta da tenere in conto il fatto che si stia comunque operando con solo una coppia di consumer e producer e con un solo topic.

#### 5.4.2 Send Parallel

Questo secondo test sfrutta Apache Bench per effettuare chiamate in parallelo all'endpoint /send.

In questo caso sono state inviate un totale di 5000 richieste avendo cura di mantenere sempre 50 richieste attive in parallelo.

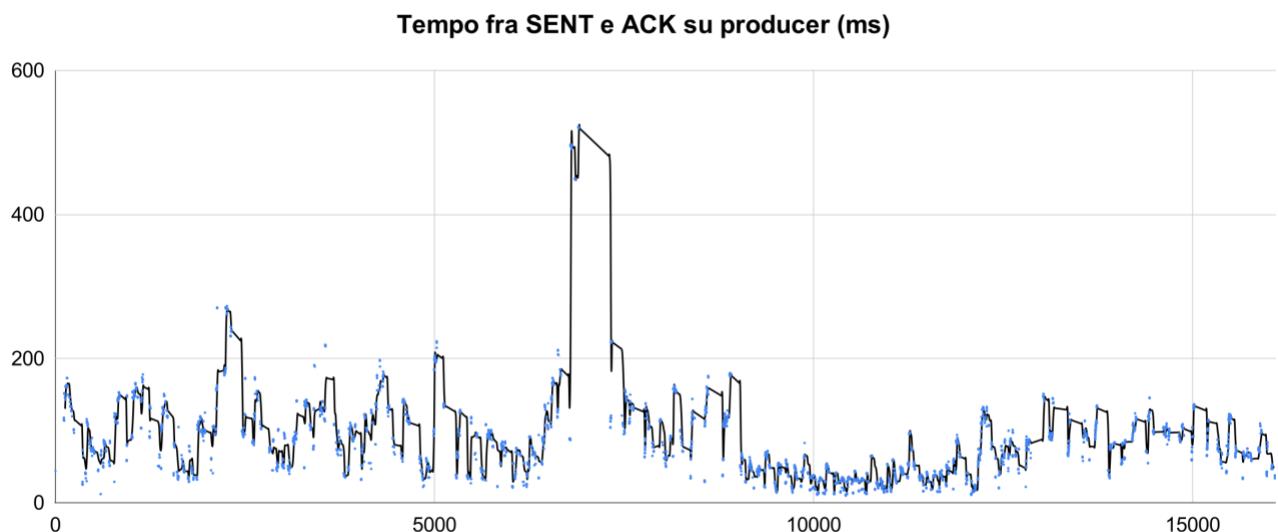


Immagine 67 - Tempo intercorso fra l'invio del messaggio al proxy e la conferma di consegna da parte di Kafka: SENT-ACK

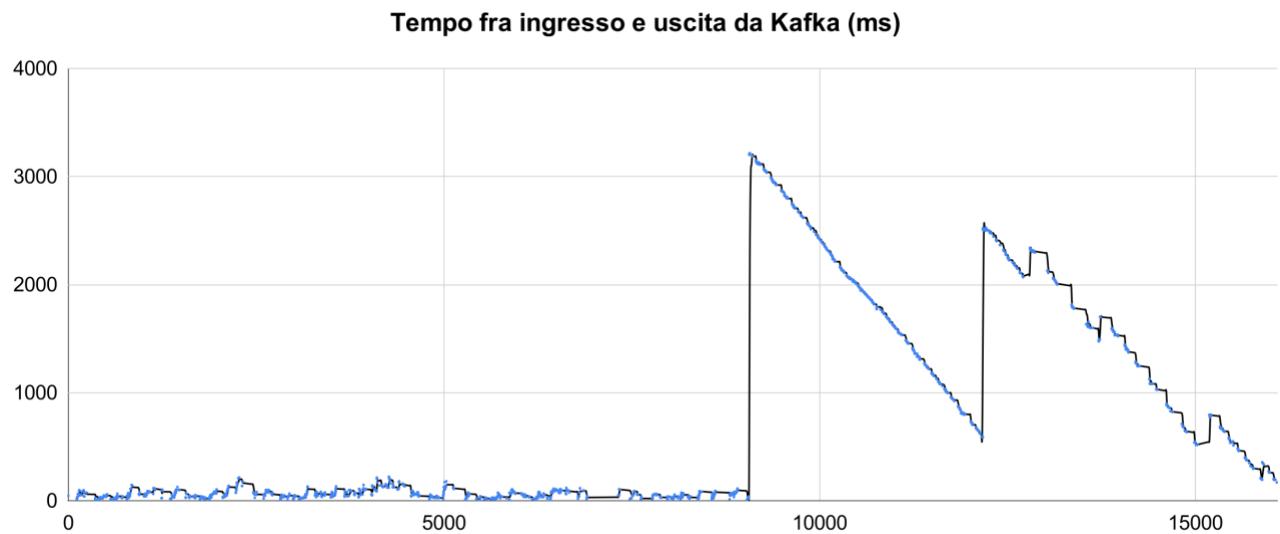


Immagine 68 - Tempo impiegato da Kafka per inoltrare il messaggio: TO\_KAFKA-FROM\_KAFKA

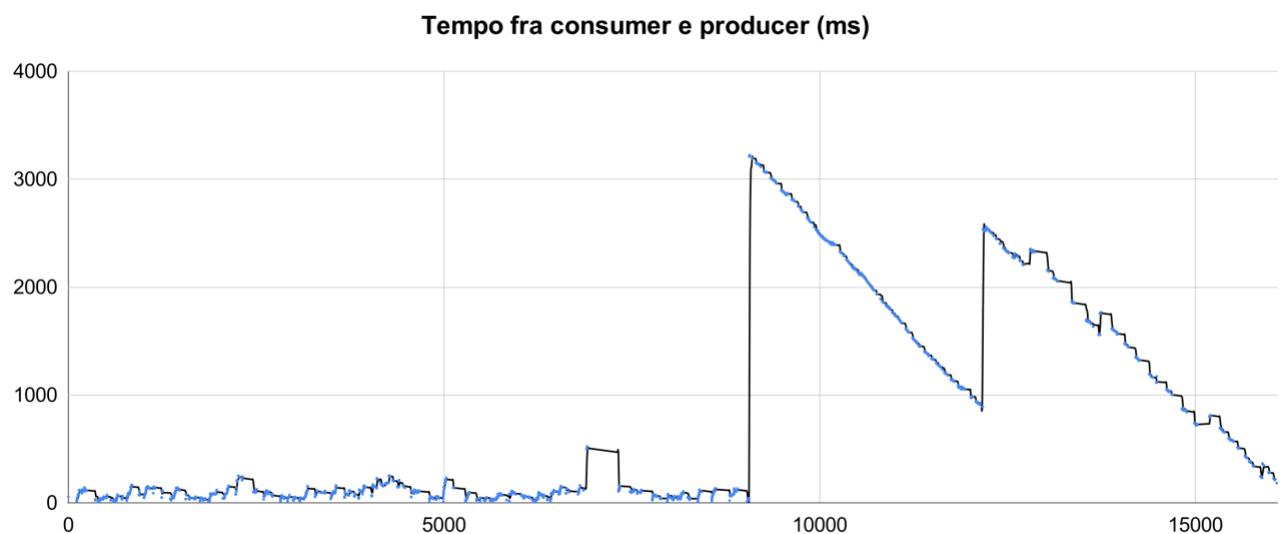
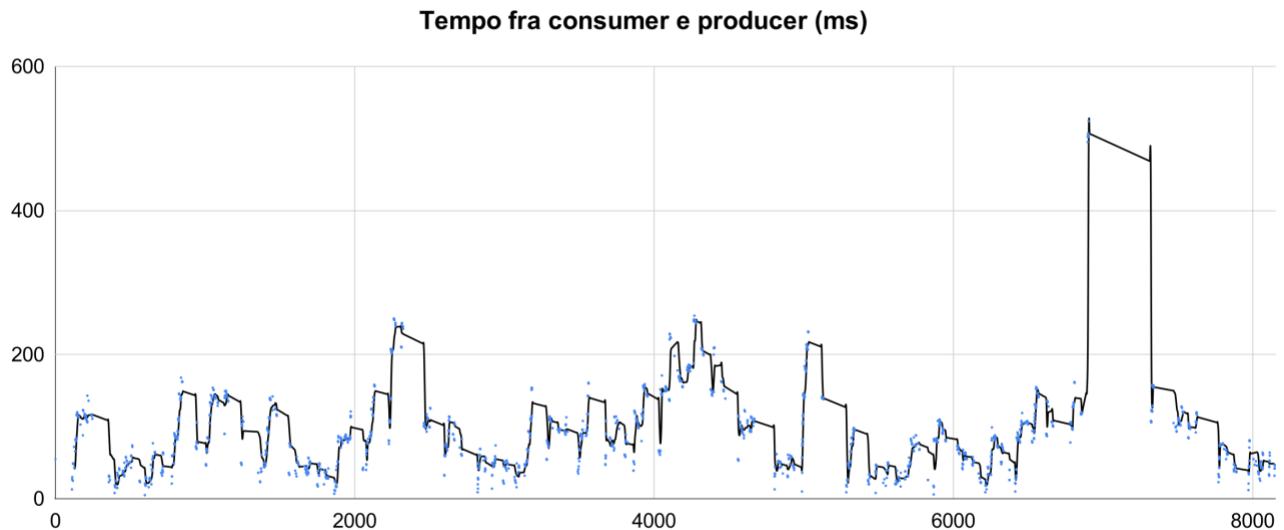


Immagine 69 - Tempo totale per la trasmissione di un messaggio: SEND-GOT

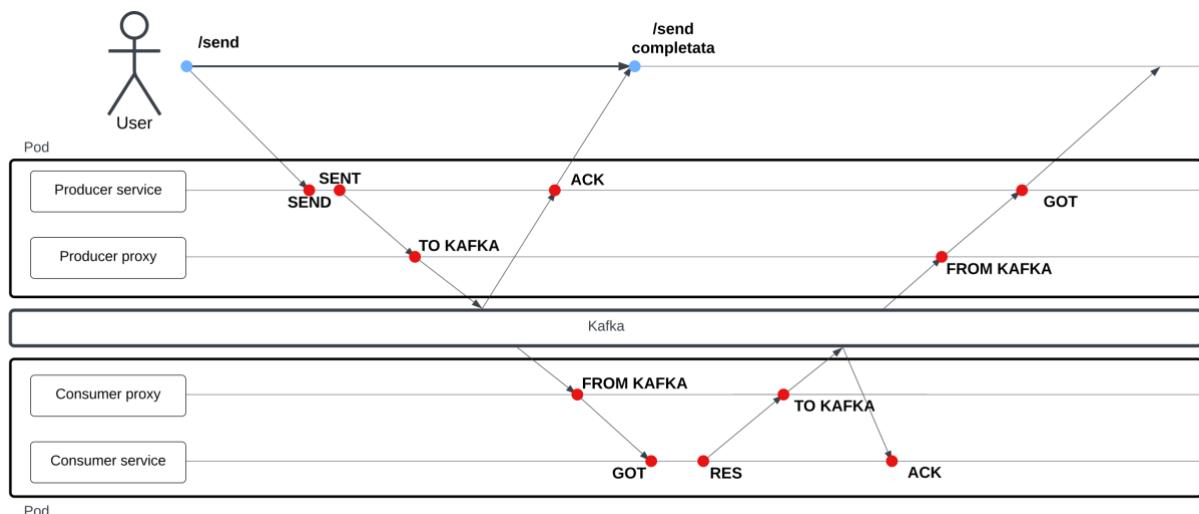


*Immagine 70 - Tempo totale per la trasmissione di un messaggio: SEND-GOT (solo i primi 8 secondi)*

Anche in questo caso è evidente come il ritardo nella trasmissione di Kafka abbia avuto un impatto drastico su tutta la comunicazione, l'entità dei rallentamenti di Kafka sono considerevoli e vanno tenuti in forte considerazione.

È da notare che 50 richieste API attive in ogni momento comportano più di 50 eventi attivi nel sistema, dal momento che una chiamata API termina quando la conferma di ricezione del proxy producer da parte di Kafka viene registrata, e non quando il consumer riceve il messaggio.

L'API termina la chiamata all'evento ACK, ma il messaggio e l'interazione continua fino all'evento GOT.



*Immagine 71 - Durata della chiamata a /send*

La media della durata della comunicazione da producer a consumer è di 88ms se si considerano gli eventi avvenuti nei primi 6 secondi.

Come si vede dal grafico, oltre a un picco contenuto causato da ritardi nel Pod producer, sono presenti due importanti rallentamenti in Kafka che hanno avuto effetti rilevanti sul risultato finale.

Nel complesso l'implementazione realizzata ha garantito risultati migliori e una maggior affidabilità del sistema rispetto al deployment Apache Event Mesh effettuato da Cultrera (Cultrera, 2022) (è stata citata questa fonte alla luce della similarità dei test effettuati).

## 5.5 Deployment Apache EventMesh

Durante la stesura di questa tesi è stato affrontato un deployment di Apache EventMesh, dopo molteplici esperimenti e molto tempo speso nel tentativo di ottenere un deployment funzionante è stata presa la decisione di rinunciare al deployment in questione e passare direttamente allo sviluppo di una implementazione alternativa.

Sebbene Apache EventMesh si presenti come un prodotto interessante è necessario tenere in considerazione che si tratta di un progetto ancora nella sua fase iniziale, soprattutto dal punto di vista dell'implementazione.

I problemi principali però sono stati causati da una documentazione incompleta e spesso errata, nessun supporto da parte della community e nessuna risorsa pubblica su blog o forum.

Molte funzionalità non sono implementate, e sebbene dalla documentazione non sembri così, addentrandosi nella repository<sup>1</sup> su GitHub e analizzando gli Issues diventa evidente che il progetto implementa una piccola frazione di quello che in realtà presenta.

Al momento AEM offre SDK solo per Java e connettori solo per RocketMQ.

Altro aspetto rilevante è dato dal fatto che AEM non offre alcun tipo di implementazione per la logica interna del proxy, è quindi necessario scrivere il codice per gestire le comunicazioni che transitano per il proxy.

Una considerazione simile è stata raggiunta anche durante la tesi magistrale del Ing. Francesco Maria Cultrera intitolata “A performance analysis of mesh models for cloud-based workflows” che descrive Apache EventMesh come uno strumento immaturo e con forti limitazioni (Cultrera, 2022), in seguito anche a test con risultati, in termini di performance, non considerati soddisfacenti.

## Conclusioni

L’impiego del cloud computing, da circa 15 anni a questa parte, risulta essere sempre più capillare in organizzazioni di qualsiasi dimensione. Tramite il cloud computing è possibile ottenere servizi offerti dai cloud provider attraverso internet, i vari tipi di servizi offerti permettono di facilitare il processo di sviluppo, supporto e manutenzione di applicativi software.

La crescita degli applicativi software in termini di importanza e complessità ha fatto sorgere diverse problematiche legate a scalabilità, fault-tolerance e modularità di architetture monolitiche, le quali hanno rappresentato lo standard architetturale de facto per decenni.

Un approccio architetturale per cercare di risolvere queste ultime problematiche, sorte con la crescita di applicativi monolitici di elevate dimensioni, è la organizzazione a microservizi che è estremamente promettente per la sua amplissima e velocissima diffusione. Tramite i microservizi, infatti, è possibile suddividere le componenti logiche delle applicazioni e trattarle come piccole unità logiche che offrono servizi in modo disaccoppiato e indipendente dal resto delle componenti appartenenti all’applicazione. L’impiego di microservizi è un cambio di paradigma necessario per garantire ad applicativi di considerevoli dimensioni di scalare, non solo in termini di performance ma anche in termini di manutenibilità, affidabilità e agilità dello sviluppo.

L’utilizzo dei microservizi aggiunge però un significativo overhead dovuto alla gestione dei numerosissimi componenti che vanno coordinati. Gli strumenti fondamentali per gestire e controllare la complessità introdotta sono gli orchestratori: fra i vari orchestratori presenti sul mercato Kubernetes è sicuramente il principale. Kubernetes permette di gestire anche applicazioni a microservizi in una varietà di ambienti diversi, partendo da architetture a singolo host sino ad arrivare a complessi cluster ospitati in cloud. Tramite molteplici astrazioni Kubernetes permette di impiegare livelli di oggetti sempre più immutabili che astraggono parti e risorse degli applicativi. Questo permette di rendere le applicazioni distribuite, resistenti ai guasti, scalabili e la cui organizzazione resti ben definita tramite il codice, grazie al concetto di Infrastructure as a Code (IaaC), implementato grazie alle configurazioni YAML.

Architetture a microservizi particolarmente imponenti in termini di numero di componenti richiedono una gestione particolarmente complessa delle interazioni fra microservizi. La soluzione analizzata per risolvere queste problematiche è ServiceMesh che rappresenta un supporto infrastrutturale dedicato alla gestione e alla comunicazione fra microservizi. Grazie a questo livello di controllo è possibile gestire le interazioni fra i microservizi in modo separato, consentendo di separare la logica dell'applicazione dalla interazione fra componenti e la loro gestione. ServiceMesh è una tecnologia sviluppata per essere fortemente scalabile e supporta modalità di deployment che ne permettono l'utilizzo in cluster distribuiti e formati da molti nodi, garantendo resistenza ai guasti e continuità di servizio.

Alla luce dei grandi vantaggi garantiti dalle tecnologie ad eventi e dai sempre maggiori impieghi che si fanno di questo approccio, si è sviluppata una soluzione che permette di facilitare l'interconnessione di servizi usando come tramite gli eventi.

EventMesh è un livello infrastrutturale dinamico, usato per mettere in comunicazione servizi e microservizi tramite la logica ad eventi. Possiede similarità con ServiceMesh, dal momento che entrambi semplificano il processo di interconnessione fra i partecipanti alla mesh. Uno dei principali vantaggi di EventMesh è quello di poter collegare applicazioni distribuite in datacenter e cloud diversi, oltre a realizzare la comunicazione ad eventi, garantendo quindi latenze basse e throughput elevati in comunicazione.

A differenza di ServiceMesh, EventMesh non offre al momento un grande numero di implementazioni: l'implementazione su cui questa tesi si concentra è Apache EventMesh.

Dopo molteplici test e tentativi di deployment non è stato possibile realizzare un deployment di Apache EventMesh funzionante. La documentazione fornita e le specifiche pubblicate in termini di supporto a strumenti esterni non sono sempre corretti, l'assenza inoltre di una procedura di deployment automatizzata complica di molto l'adozione di questo strumento. Al momento Apache EventMesh non sembra essere ancora un'implementazione matura per essere impiegata in ambienti di produzione, sebbene le premesse del progetto siano buone, al momento questa implementazione è acerba e difficilmente utilizzabile in scenari reali.

Di contro, l'implementazione all'interno di questa tesi, seppur con tecnologie e approcci leggermente diversi da quelli impiegati da Apache EventMesh ha prodotto interessanti risultati soprattutto in termini di tempi di latenza. L'implementazione è stata realizzata attraverso sidecar proxy realizzati in NodeJS, i quali comunicano tramite WebSocket con i microservizi e via SDK con Kafka, impiegato come event broker. L'impiego di NodeJS permette l'utilizzo di molte valide librerie che garantiscono uno sviluppo rapido di eventuali integrazioni con altri MOM o altri sistemi esterni.

Gli sviluppi futuri di questa implementazione dovrebbero concentrarsi sulla modularità del codice e nel rendere più netta la distinzione fra logica integrata nel proxy e logica integrata nel microservizio, la soluzione più efficace sarebbe fornire connettori microservizio-proxy da usare attraverso funzioni di libreria. Dal momento che Kafka, come visto dai risultati dei benchmark, ha un impatto così rilevante sulle performance di tutta la organizzazione Event Mesh, sarebbe anche opportuno migliorare l'integrazione con Kafka e rivedere il deployment Kafka per ottimizzarlo il più possibile per questo caso d'uso particolare.

Nel complesso una tecnologia come Event Mesh è particolarmente interessante soprattutto alla luce dell'ampio impiego che le aziende stanno iniziando a fare delle architetture ad eventi. Al momento, non si tratta di una tecnologia tanto matura quanto Service Mesh ma resta senza dubbio un ottimo punto di partenza per la creazione di layer infrastrutturali dinamici e orientati agli eventi.

## Bibliografia

- IBM. (s.d.). *What is cloud computing?* Tratto da <https://www.ibm.com/topics/cloud-computing>
- Varghese, B. (2019, Marzo 2019). *History of the cloud.* Tratto da BCS: <https://www.bcs.org/articles-opinion-and-research/history-of-the-cloud/>
- Garfinkel, S. (2011, Ottobre 3). *The Cloud Imperative.* Tratto da MIT Technology Review: <https://www.technologyreview.com/2011/10/03/190237/the-cloud-imperative/>
- Arrington, M. (2008, Aprile 8). *Google Jumps Head First Into Web Services With Google App Engine.* Tratto da Tech Crunch: <https://techcrunch.com/2008/04/07/google-jumps-head-first-into-web-services-with-google-app-engine/>
- MSV, J. (2020, Febbraio 3). *A Look Back At Ten Years Of Microsoft Azure.* Tratto da Forbes: <https://www.forbes.com/sites/janakirammsv/2020/02/03/a-look-back-at-ten-years-of-microsoft-azure/>
- Fisher, C. (2018, Settembre). Cloud versus On-Premise Computing. *American Journal of Industrial and Business Management*, 8.
- Synergy Research Group. (2023, Febbraio 6). *Cloud Spending Growth Rate Slows But Q4 Still Up By \$10 Billion from 2021; Microsoft Gains Market Share.* Tratto da Synergy research group: <https://www.srgresearch.com/articles/cloud-spending-growth-rate-slows-but-q4-still-up-by-10-billion-from-2021-microsoft-gains-market-share>
- Google Cloud. (s.d.). *What is a Public Cloud?* . Tratto da Google Cloud Learn: <https://cloud.google.com/learn/what-is-public-cloud>
- Microsoft Azure. (s.d.). *What is a private cloud?* Tratto da Microsoft Azure: <https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-private-cloud>
- IBM. (s.d.). *What is private cloud?* Tratto da IBM: <https://www.ibm.com/topics/private-cloud>
- Google Cloud. (s.d.). *What is a Hybrid Cloud?* Tratto da Google Cloud Learn: <https://cloud.google.com/learn/what-is-hybrid-cloud>
- Gartner. (2022, Ottobre 31). *Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly \$600 Billion in 2023.* Tratto da Gartner: <https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023>
- Google Cloud. (s.d.). *Google Cloud learn.* Tratto da What is Platform as a Service (PaaS)?: <https://cloud.google.com/learn/what-is-paas>
- Cloudflare. (s.d.). *What is Function-as-a-Service?* Tratto da Cloudflare: <https://www.cloudflare.com/it-it/learning/serverless/glossary/function-as-a-service-faas/>
- IBM. (s.d.). *What is virtualization?* Tratto da IBM: <https://www.ibm.com/topics/virtualization>

IBM. (s.d.). *What are hypervisors?* Tratto da IBM: <https://www.ibm.com/topics/hypervisors>

Google Cloud. (s.d.). *What is Microservices Architecture?* Tratto da Google Cloud Learn: <https://cloud.google.com/learn/what-is-microservices-architecture>

IBM. (s.d.). *What are microservices?* Tratto da IBM: <https://www.ibm.com/topics/microservices>

Amanse, A. (s.d.). *Why should you use microservices and containers?* Tratto da IBM: <https://developer.ibm.com/articles/why-should-we-use-microservices-and-containers/>

RedHat. (2021, Aprile 8). *Cos'è la containerizzazione?* Tratto da RedHat: <https://www.redhat.com/it/topics/cloud-native-apps/what-is-containerization>

Docker. (s.d.). *Use containers to Build, Share and Run your applications.* Tratto da Docker: <https://www.docker.com/resources/what-container/>

RedHat. (2019, Ottobre 15). *Cos'è l'orchestrazione?* Tratto da RedHat: <https://www.redhat.com/it/topics/automation/what-is-orchestration>

Docker. (s.d.). *Deployment and orchestration.* Tratto da Docker Docs: <https://docs.docker.com/get-started/orchestration/>

Cultrera, F. M. (2022). *A performance analysis of mesh models for cloud-based workflows.*

Docker. (s.d.). *Docker overview.* Tratto da Docker Docs: <https://docs.docker.com/get-started/overview/>

Docker . (s.d.). *The Industry-Leading Container Runtime.* Tratto da Docker: <https://www.docker.com/products/container-runtime/>

Microsoft Learn. (2022, Aprile 4). *Docker containers, images, and registries.* Tratto da Microsoft Learn: <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-containers-images-registries>

Docker. (s.d.). *Swarm mode key concepts.* Tratto da Docker Docs: <https://docs.docker.com/engine/swarm/key-concepts/>

Docker . (s.d.). *How nodes work.* Tratto da Docker Docs: <https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/>

<https://kubernetes.io/docs/concepts/overview/>. (s.d.). *Overview.* (Kubernetes) Tratto il giorno Maggio 17, 2023 da <https://kubernetes.io/docs/concepts/overview/>

Kubernetes. (s.d.). *Kubernetes Components.* Tratto il giorno May 17, 2023 da <https://kubernetes.io/docs/concepts/overview/components/>

Kubernetes. (2022, Gennaio 8). *Controllers.* Tratto da <https://kubernetes.io/docs/concepts/architecture/controller/>

Kubernetes. (2023, Aprile 13). *Scheduling Framework.* Tratto da <https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/>

- Kubernetes. (2023, Marzo 21). *Cloud Controller Manager*. Tratto da <https://kubernetes.io/docs/concepts/architecture/cloud-controller/>
- Kubernetes. (2023, Maggio 16). *Objects in Kubernetes*. Tratto da <https://kubernetes.io/docs/concepts/overview/working-with-objects/>
- Kubernetes. (2023, Marzo 28). *Pods*. Tratto da <https://kubernetes.io/docs/concepts/workloads/pods/>
- Kubernetes. (2023, May 23). *Deployments*. Tratto da <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>
- Tremel, E. (2017, Settembre 25). *Kubernetes deployment strategies*. Tratto da <https://blog.container-solutions.com/kubernetes-deployment-strategies>
- Kubernetes. (2023, Marzo 25). *Using a Service to Expose Your App*. Tratto da <https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/>
- Patel, A. (2021, Giugno 1). *Kubernetes — Service Types Overview*. (DevOps Mojo) Tratto da <https://medium.com/devops-mojo/kubernetes-service-types-overview-introduction-to-k8s-service-types-what-are-types-of-kubernetes-services-ea6db72c3f8c>
- Kubernetes. (2022, Novembre 4). *Services, Load Balancing, and Networking*. Tratto da <https://kubernetes.io/docs/concepts/services-networking/>
- Dancuk, M. (2022, Giugno 7). *Kubernetes Networking Guide*. (PhoenixNAP) Tratto da <https://phoenixnap.com/kb/kubernetes-networking>
- Sookocheff, K. (2018, Luglio 11). *A Guide to the Kubernetes Networking Model*. Tratto da <https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/>
- How it Works*. (s.d.). (K3S) Tratto da <https://k3s.io/>
- Donston-Miller, D. (2021, Marzo 11). *Service mesh: why and when to use one with microservices*. (RedHat) Tratto da <https://www.redhat.com/architect/why-when-service-mesh>
- Velichko, I. (2021, Agosto 7). *Sidecar Proxy Pattern - The Basis Of Service Mesh*. Tratto da <https://iximiuz.com/en/posts/service-proxy-pod-sidecar-oh-my/>
- O'Reilly. (s.d.). *Chapter 2. The Sidecar Pattern*. (O'Reilly) Tratto il giorno Giugno 26, 2023 da <https://www.oreilly.com/library/view/designing-distributed-systems/9781491983638/ch02.html>
- VMWare. (s.d.). *What is a Service Mesh?* (VMWare) Tratto il giorno Giugno 26, 2023 da <https://www.vmware.com/topics/glossary/content/service-mesh.html>
- Kleeman, J. (2019, Novembre 6). *We built network isolation for 1,500 services to make Monzo more secure*. (Monzo) Tratto da <https://monzo.com/blog/we-built-network-isolation-for-1-500-services>
- Google Cloud. (2021, Febbraio 16). *Mesh di servizi in un'architettura di microservizi*. (Google Cloud) Tratto da <https://cloud.google.com/architecture/service-meshes-in-microservices-architecture?hl=it>

O'Reilly. (s.d.). *Chapter 1. Service Mesh Fundamentals.* (O'Reilly) Tratto da <https://www.oreilly.com/library/view/the-enterprise-path/9781492041795/ch01.html>

Istio. (s.d.). *Architecture.* (Istio) Tratto da <https://istio.io/latest/docs/ops/deployment/architecture/>

Istio. (s.d.). *Pilot.* (Istio) Tratto da <https://istio.io/v0.4/docs/concepts/traffic-management/pilot.html>

Istio. (s.d.). *Traffic management benefits.* (Istio) Tratto da <https://istio.io/v0.4/docs/concepts/traffic-management/overview.html>

Istio. (s.d.). *Galley.* (Istio) Tratto da <https://istio.io/v1.4/docs/ops/deployment/architecture/#envoy>

Varga, Z. (2020, Aprile 12). *Istio telemetry V2 (Mixerless) deep dive.* (Banzai Cloud) Tratto da <https://banzaicloud.com/blog/istio-mixerless-telemetry/>

Istio. (s.d.). *Deployment Models.* (Istio) Tratto da <https://istio.io/latest/docs/ops/deployment/deployment-models/>

RedHat. (Agosto, 19). *Che cos'è un event mesh?* . (2021) Tratto da <https://www.redhat.com/it/topics/integration/what-is-an-event-mesh>

Menning, J. (2022, Maggio 10). *How a Sidecar Proxy Can Rule your Event Mesh.* (Solace) Tratto da <https://solace.com/blog/sidecar-proxy-rule-event-mesh/>

WeBank. (2023, Giugno 8). *In-Depth Analysis of Apache EventMesh: A Framework for Distributed Application Efficiency.* (WeBank) Tratto da <https://medium.com/@webankcoretech/in-depth-analysis-of-apache-eventmesh-a-framework-for-distributed-application-efficiency-bad8d3c00156>

EventMesh. (s.d.). *EventMesh Schema Registry (OpenSchema)* . (EventMesh) Tratto da <https://eventmesh.apache.org/docs/design-document/schema-registry>

Strothmann, K. (2023, Febbraio 1). *SAP Event Mesh: Trial to be retired.* (SAP) Tratto da <https://blogs.sap.com/2023/02/01/sap-event-mesh-trial-to-be-retired/>

Apache EventMesh. (s.d.). *EventMesh Runtime Protocol.* (Apache EventMesh) Tratto da <https://eventmesh.apache.org/docs/design-document/runtime-protocol>

Apache EventMesh. (s.d.). *EventMesh Schema Registry (OpenSchema).* (Apache EventMesh) Tratto da <https://eventmesh.apache.org/docs/design-document/schema-registry>

MacKinnon, G. (2019, Settembre 2019). *#2 Service mesh origins (a little history).* Tratto da <https://medium.com/@fern crusher/2-service-mesh-origins-a-little-history-99b8506666b4>

Braun, D. (2022, Dicembre 8). *Service Meshes for Kubernetes: Unlocking Standardized Security, Resilience, and Traffic Management.* (Semaphore) Tratto da <https://semaphoreci.com/blog/service-meshes-kubernetes>

asds. (adsf). ads. *sd, asdf(sdf).*

AWS. (s.d.). *Cos'è l'EDA?* (AWS) Tratto da [https://aws.amazon.com/it/what-is/eda/?nc1=h\\_ls](https://aws.amazon.com/it/what-is/eda/?nc1=h_ls)

Istio. (s.d.). *Istio.* (Istio) Tratto da <https://istio.io/latest/>

AWS. (s.d.). *AWS App Mesh*. (AWS) Tratto da <https://aws.amazon.com/it/app-mesh/>

Linkerd. (s.d.). *Linkerd*. (Linkerd) Tratto da <https://linkerd.io/>

HashiCorp. (s.d.). *Consul*. (HashiCorp Consul) Tratto da <https://www.consul.io/>

AWS. (s.d.). *Amazon CloudWatch*. Tratto da Amazon AWS: <https://aws.amazon.com/it/cloudwatch/>

RedHat. (s.d.). *redHat Openshift*. Tratto da Redhat: <https://www.redhat.com/en/technologies/cloud-computing/openshift>