



Corso di Laurea Magistrale in Informatica

Anno Accademico 2018/2019

Software Architecture

Kubernetes

Matteo Marchesini

0000856336

matteo.marchesini12@studio.unibo.it

Indice

1	Introduzione	3
1.1	Cos'è la Container Orchestration ?	3
1.2	Descrizione del sistema	4
2	Descrizione dell'architettura	5
2.1	Scopo	5
2.2	Contesto	6
2.2.1	Stakeholders	7
2.2.2	Development	9
2.2.3	Piattaforme	11
3	Drivers architetturali	12
3.1	Scalabilità	12
3.2	Disponibilità	14
3.3	Sicurezza	15
3.4	Portabilità	18
4	Struttura architetturale	19
4.1	Pod	20
4.2	Service (SOA)	21
4.3	Nodo Master	25
4.3.1	API Server	26
4.3.2	etcd-storage	26
4.3.3	Scheduler	26
4.3.4	Controller	27
4.4	Command Line Interface	28

4.5	Node	29
4.5.1	kubelet	30
4.5.2	kube-proxy	30
4.5.3	Container runtime	31
4.5.4	Logging Layer	31
5	Deployment patterns	32
6	Aspetti analitici e criticità	35
6.1	Code Quality	35
6.2	Utility tree	36
7	Stili architetturali simili: Docker Swarm	39

Capitolo 1

Introduzione

1.1 Cos'è la Container Orchestration ?

Un **container** è una tecnologia per il deployment di sistemi distribuiti, una forma di server virtualizzato a livello di sistema operativo. Esso è un insieme di processi isolati dal resto del sistema, che eseguono un'immagine distinta contenente tutti i file necessari per supportare tali processi. Offrendo un'immagine che contiene tutte le dipendenze di un'applicazione, la containerizzazione offre portabilità e coerenza nel passaggio dallo sviluppo al testing, fino alla produzione.

Come fare quando esistono più container ?

È necessaria un'**orchestrazione di container**.

L'orchestrazione di container è un processo che permette di distribuire una molteplicità di container per poter implementare un'applicazione. Infatti nel development moderno le applicazioni non sono più monolitiche, ma sono invece composte da dozzine o centinaia di componenti containerizzati liberamente accoppiati che devono lavorare insieme per consentire a una determinata app di funzionare come progettata. L'orchestrazione dei container fa riferimento al processo di organizzazione del lavoro dei singoli componenti e dei livelli dell'applicazione.

Tool di orchestrazione di container, quali Kubernetes, Apache Mesos e Docker Swarm, consentono agli utenti di guidare il deployment, gli aggiornamenti automatici e il monitoraggio di container.

1.2 Descrizione del sistema



kubernetes

Kubernetes è un sistema open source per la gestione di applicazioni containerizzate tra più host; fornisce un meccanismo per il deployment, la manutenzione e lo scaling di applicazioni. È stato inizialmente sviluppato dal team di Google, per poi passare nel 2015 sotto il controllo del *Cloud Native Computing Foundation (CNCF)*, che attualmente lo supporta.

Al giorno d'oggi è uno dei sistemi di orchestrazione per applicazioni containerizzate più utilizzato in assoluto, con una vastità di utenti, partners e una comunità di development attiva. Non a caso tre dei quattro maggior providers di servizi Cloud - Microsoft, IBM e Google - offrono piattaforme di Container as a Service (CaaS) basate su Kubernetes. I servizi che Kubernetes mette a disposizione sono molteplici: fornisce un ambiente la gestione di container, microservizi e piattaforme cloud. Inoltre organizza l'infrastruttura di rete e di archiviazione per conto dell'utente.

In generale un sistema distribuito necessita di più componenti per il corretto funzionamento, alcune open source e altre commerciali; invece Kubernetes da solo fornisce uno scenario in cui le componenti lavorano insieme, andando così a formare un unico componente combinando la semplicità del Platform as a Service (PaaS) con la flessibilità dell'Infrastructure as a Service (IaaS).

L'orchestrazione di container ha avuto un profondo impatto in ogni aspetto del software development e deployment moderno; in particolare ha influenzato l'architettura del Platform as a Service, fornendo un aperto ed efficiente modello per il packaging, deployment, isolamento, scaling e rolling upgrade. Kubernetes svolge un ruolo cruciale nell'utilizzo di container da parte di imprese e start-up emergenti.

L'oggetto di questo report è di studiare tutto ciò che riguarda e circonda l'architettura di Kubernetes.

Capitolo 2

Descrizione dell'architettura

2.1 Scopo

In questo capitolo verrà discusso lo scopo di Kubernetes e la sua interazione con le entità esterne, nonché i principali casi d'uso.

Kubernetes, come introdotto nel Capitolo 1, è un sistema di orchestrazione di container e per questo motivo si occupa principalmente di **deployment**, **scaling** e **management** di applicazioni containerizzate. Di seguito viene definito ognuno di questi tasks:

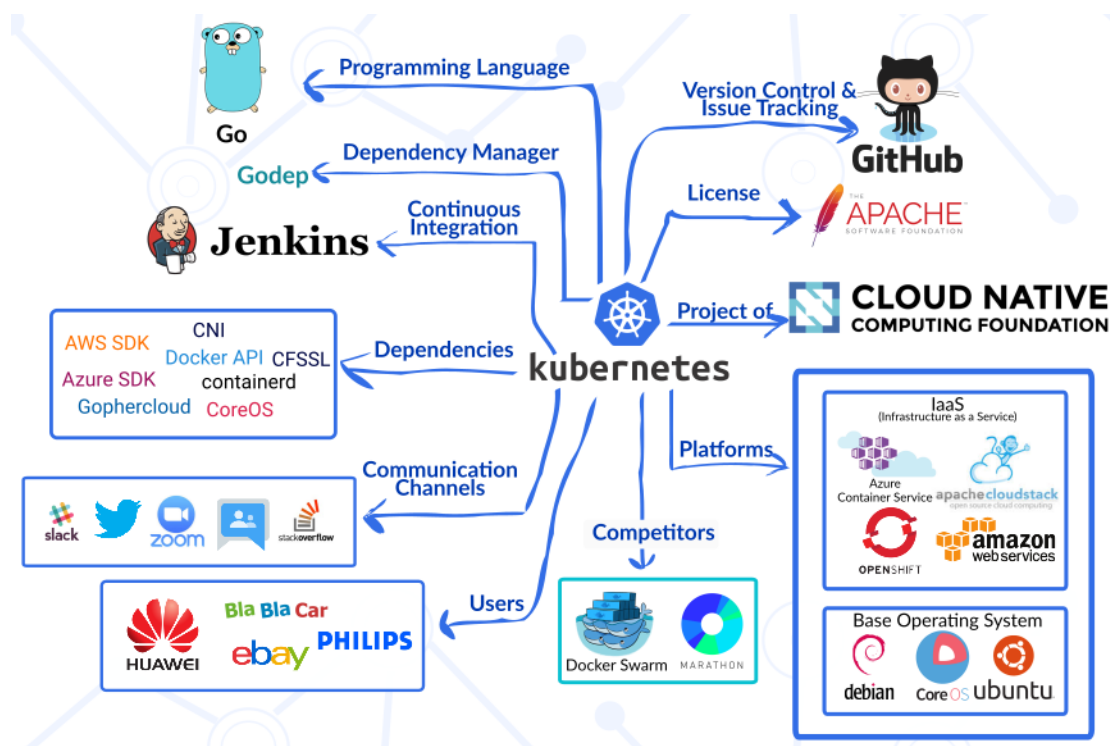
- **Deployment:** gestisce la distribuzione di applicazioni assegnando ai nodi del cluster ciascuna istanza dell'applicazione. Il deployment in Kubernetes può essere eseguito in una varietà di ambienti con pattern differenti, ed esistono appunto diversi modelli, quali:
 - Container as a Service (CaaS);
 - Platform as a Service (PaaS);
 - Utilizzo on-premises all'interno di data center;
 - Deployment personalizzato
- **Scaling:** permette di ridimensionare l'applicazione a seconda delle esigenze dell'utente, andando a modificare le dimensioni del cluster e il numero di repliche dei pod. Un **pod** è il più piccolo oggetto deployabile nel mo-

dello a oggetti di Kubernetes e può incapsulare un singolo container o più container che necessitano di lavorare insieme.

- **Management:** fornisce un'interfaccia per la gestione dei cluster e delle applicazioni containerizzate. Un cluster di Kubernetes viene ospitato e gestito da un venditore commerciale, quali ad esempio Google Container Engine (GKE). Molti utenti hanno iniziato ad usare Kubernetes attraverso Google Container Engine, essendo uno dei primi servizi di gestione di Kubernetes nel mercato.

2.2 Contesto

Nella figura seguente è espresso il *system context diagram* di Kubernetes, che definisce le relazioni tra esso e le entità esterne.



Le entità di maggior rilievo all'interno del diagramma e che verranno analizzate in seguito sono gli stakeholders, il development, le piattaforme e i concorrenti.

2.2.1 Stakeholders

Per poter comprendere al meglio il concetto di stakeholder all'interno di un progetto così grande come quello di Kubernetes, è necessario introdurre il **Cloud Native Computing Foundation (CNCF)**.

CNCF è nato nel 2015 da un accordo tra Google e la Linux Foundation con l'annuncio di Kubernetes 1.0, considerato il progetto principale. Da lì in poi molte industrie del cloud computing si sono unite a CNCF per incubare, sviluppare e mantenere un ecosistema di progetti cloud sotto una visione comune e condivisa. I membri di CNCF sono divisi per categorie, quali Platinum, Gold, Silver, End-User, accademici e no-profit. Tra i membri Platinum abbiamo Google, Docker, IBM, Cisco e Oracle.

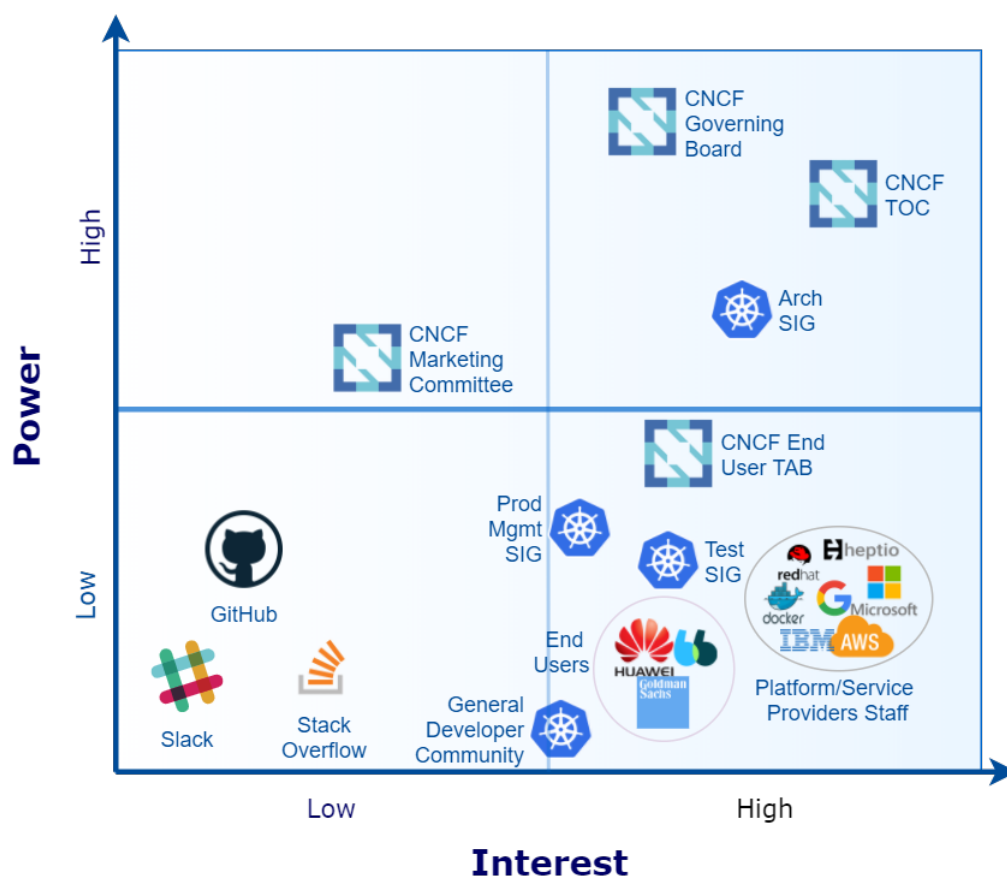
Oltre all'organizzazione CNCF, Kubernetes attrae migliaia di contributori che coordinano i loro sforzi attraverso piattaforme online, quali GitHub, Slack e StackOverflow (fornitori). Gli Special Interest Groups (SIG) si occupano dello sviluppo di Kubernetes, in particolare dell'architettura, del product management e del testing, nonché di implementazione specifiche per fornitori di servizi come AWS e Azure.

Griglia Power/Interest

Nella seguente immagine è raffigurata la griglia o matrice *power/interest* che divide gli stakeholder in quattro categorie:

- **Alta potenza e alto interesse:** la massima potenza sui progetti di Kubernetes la detiene il consiglio d'amministrazione di CNCF che ne gestisce il budget, seguito poi da CNCF TOC (Technical Oversight Committee), che ha il compito di aggiungere e rimuovere progetti come Kubernetes. Inoltre anche l'architettura SIG è fondamentale in quanto decide il futuro del progetto.
- **Alta potenza e basso interesse:** il CNCF Marketing Committee, derivato dal consiglio di amministrazione di CNCF, cura il brand del progetto e altre attività legate al business.

- **Bassa potenza e alto interesse:** rispetto agli stakeholder con grande potere su Kubernetes, il CNCF end user TAB, lo staff di provider di piattaforme/servizi, Test SIG e la comunità di sviluppatori generali di Kubernetes dipendono dal continuo sviluppo del progetto, e questo comporta un alto interesse.
- **Bassa potenza e basso interesse:** le piattaforme di coordinamento, quali GitHub, hanno scarso potere ed interesse per il progetto, in quanto altre piattaforme potrebbero avere uno scopo simile se non fossero prese in considerazione.



2.2.2 Development

Il principale linguaggio con cui Kubernetes è sviluppato è **Go**, un linguaggio open-source progettato da tre ingegneri di Google, ma nonostante ciò possiede dipendenze da varie librerie esterne, quali Amazon Web Services SDK (AWS), Docker API, Azure SDK, Container Network Interface (CNI), e Gophercloud. Inoltre Kubernetes si affida a Jenkins per la Continuous Integration.

Organizzazione a moduli

Il codice sorgente di Kubernetes è organizzato in livelli di moduli in base alle funzionalità. La panoramica generale della partizione dei moduli è mostrata nella Figura 2.1.

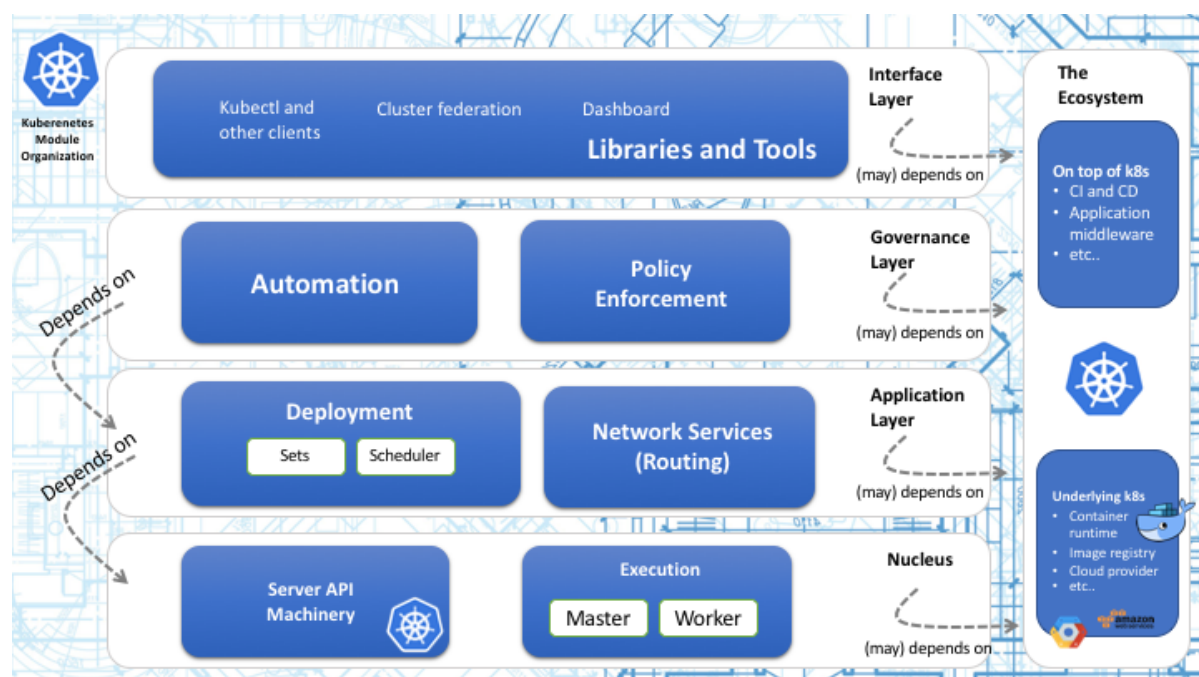


Figura 2.1: Organizzazione a moduli di Kubernetes

Di seguito viene spiegato ciascun livello.

- **Nucleo - API ed Esecuzione**

Il nucleo contiene il set minimo di funzionalità necessarie per costruire i livelli superiori del sistema ed è composto da moduli API ed Execution. I moduli API forniscono una raccolta di API REST e i moduli di esecuzione di Kubernetes sono responsabili dell'esecuzione dell'applicazione all'interno dei container. Il modulo di esecuzione più importante in Kubernetes è *Kubelet*.

- **Livello Applicazione - Deployment e Routing**

Il livello dell'applicazione fornisce auto-healing, scaling, gestione del ciclo di vita delle applicazioni, individuazione dei servizi, bilanciamento del carico e routing. Inoltre, il modulo di deployment fornisce metodi container-centric e controller del ciclo di vita per supportare l'orchestrazione, mentre il modulo di routing fornisce il servizio di scheduling.

- **Livello Governance - Automazione e Policy Enforcement**

Questo livello contiene il modulo di automazione che fornisce lo scaling automatico del cluster e il provisioning automatico dei nodi. Il modulo di policy enforcement fornisce i mezzi per configurare e scoprire le politiche predefinite per il cluster.

- **Livello Interface - Librerie e Tools**

Questo livello contiene librerie, tools, sistemi e interfacce utente comunemente usate nel progetto Kubernetes. Uno degli strumenti più importanti in questo livello è **kubectl**. Contiene anche altre librerie client come *client-go* e *client-python*.

- **Ecosistema**

Questo livello non fa realmente parte di Kubernetes, ma fornisce le funzionalità comunemente necessarie per distribuire e gestire applicazioni containerizzate. L'esempio più popolare di questo livello di supporto è Docker, che esegue il container reale nei nodi. I moduli utilizzati per comunicare con il provider di cloud appartengono a questo livello.

2.2.3 Piattaforme

Kubernetes essendo una piattaforma cloud è compatibile con diversi providers a seconda delle necessità e dell'utilizzo. Di seguito sono riportate alcune soluzioni differenti a seconda del pattern di utilizzo.

Prodotto (compagnia)	Categoria
AppCode (AppCode) <i>È una piattaforma integrata per il deployment, testing e coding di app containerizzate. Permette il deployment su Google Cloud Platform e AWS.</i>	Hosted, PaaS
Giant Swarm (Giant Swarm) <i>Una soluzione creare, distribuire e gestire servizi containerizzati con Kubernetes come componente principale</i>	Hosted, on-premises
Google Container Engine (Google) <i>Google Container Engine è un sistema di gestione e orchestrazione dei cluster che consente agli utenti di eseguire container sulla piattaforma Cloud di Google</i>	CaaS
OpenShift Online (RedHat) <i>Una piattaforma di applicazioni per container che può estendersi su più infrastrutture. È costruito usando la tecnologia Docker e Kubernetes.</i>	PaaS, on-premises
Minikube (Cloud Native Computing Foundation) <i>Una piattaforma di applicazioni per container che può estendersi su più infrastrutture. È costruito usando la tecnologia Docker e Kubernetes.</i>	Deploy
Platform9 Managed Kubernetes for Docker (Platform9) <i>I clienti possono utilizzare il singolo pannello di Platform 9 per orchestrare e gestire i container insieme alle macchine virtuali. In pratica, è possibile orchestrare le VM utilizzando OpenStack e/o Kubernetes</i>	Hosted, on-premises
Kubernetes Dashboard (Cloud Native Computing Foundation) <i>Fornisce un'interfaccia utente basata sul Web per cluster Kubernetes. Consente agli utenti di gestire le applicazioni in esecuzione nel cluster nonché di gestire il cluster stesso.</i>	Monitoring
Agile Stacks (Cloud Native Computing Foundation) <i>It provides automation to deploy Kubernetes into VPC in public cloud</i>	IaaS

Capitolo 3

Drivers architetturali

Kubernetes, come si può capire dal capitolo precedente, è progettato secondo i principi di scalabilità, disponibilità e sicurezza, nonchè fornisce una distribuzione del carico di lavoro tra le risorse disponibili. Questo capitolo metterà in evidenza alcuni dei requisiti non funzionali di Kubernetes.

3.1 Scalabilità

Le applicazioni sono eseguite in Kubernetes come microservizi. Tali microservizi sono composti da più container raggruppati in **pods**. Ciascun container è progettato per eseguire solamente un unico task.

I pods possono essere composti da container con o senza stato (stateless/stateful). I pods senza stato possono essere scalati facilmente quando necessario anche attraverso l'auto-scaling dinamico.

Dalla versione 1.4 Kubernetes supporta l'auto-scaling orizzontale dei pod, che scala automaticamente il numero di pod in repliche controllate basate sull'utilizzo della CPU. La versione di Kubernetes ospitata da Google Cloud supporta l'auto-scaling di cluster. Quando un nuovo pod viene scalato tra tutti i nodi disponibili, Kubernetes coordina l'infrastruttura sottostante per far sì che ulteriori nodi vengano aggiunti al cluster.

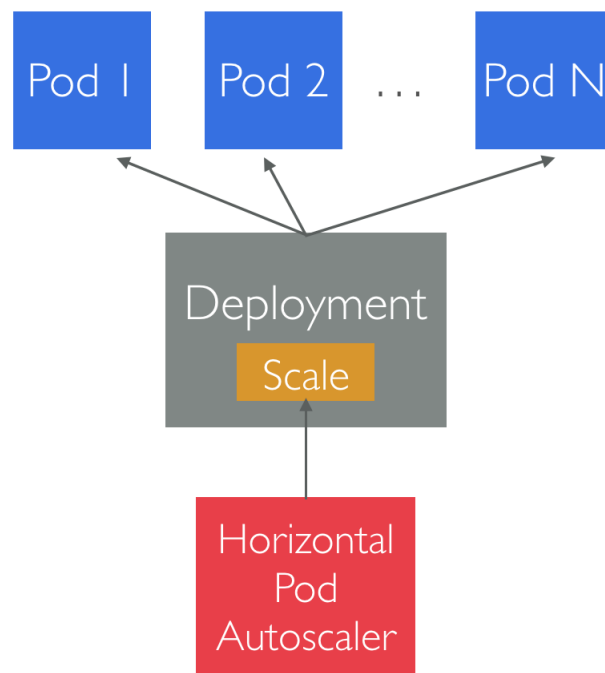


Figura 3.1: Autoscaling orizzontale di un pod

L'auto-scaling orizzontale è implementato come un loop di controllo, in cui periodicamente (con un valore predefinito di 30 secondi) il controller manager interroga l'utilizzo delle risorse rispetto alle metriche specificate in ciascuna definizione di *HorizontalPodAutoscaler*. Nello specifico, il controller manager ottiene le metriche (ad esempio l'utilizzo di CPU) attraverso l'API apposita per le metriche; quindi se è settato un valore target di utilizzo, il controller calcola il valore di utilizzo come percentuale delle richieste alle risorse equivalenti contenute nei container di ciascun pod. Se invece è impostato un valore grezzo, il controller utilizza direttamente tale valore. Il controller quindi assume o la media di utilizzo o il valore grezzo, per produrre poi un rapporto usato per scalare il numero di repliche desiderate. Da sottolineare il fatto che se alcuni container all'interno dei pods non hanno il set di richieste di risorse rilevanti, l'utilizzo di CPU per quel dato pod non verrà definito e l'autoscaler non intraprenderà alcuna azione per tale metrica. Kubernetes supporta inoltre lo scaling di applicazioni stateless come i cluster Cassandra e i set di repliche MongoDB, ovvero più in generale di workloads persistenti, quali i database NoSQL e i database relazionali (RDBMS).

3.2 Disponibilità

In kubernetes i carichi di lavoro richiedono la disponibilità sia a livello di infrastruttura che applicazione. Ad esempio, nei cluster a larga scala, tutto è soggetto ai guasti, il che rende necessaria un'elevata disponibilità per la produzione dei workloads. Mentre la maggior parte dei motori di orchestrazione di container offrono la disponibilità di applicazioni, Kubernetes è progettato per affrontare la disponibilità sia di applicazioni che di infrastrutture.

Per quanto riguarda il fronte applicazione, Kubernetes assicura un'alta disponibilità per mezzo di *ReplicaSet*, *Replication Controller* e *StatefulSets*. Un *ReplicaSet* garantisce che un numero specificato di repliche di pod siano in esecuzione in qualsiasi momento, e lo stesso vale per i *Replication Controller*, con l'unica differenza che quest'ultimo supporta solo il selettore di uguaglianza, mentre *ReplicaSet* supporta il selettore basato sul set.

Quindi l'utilizzatore di Kubernetes può dichiarare il numero minimo di pods che devono essere eseguiti in un dato istante di tempo. Se un container o un pod si arresta in modo anomalo a causa di un errore, la policy dichiarativa può riportare il deployment alla configurazione desiderata.

Analizzando la disponibilità rispetto all'infrastruttura, Kubernetes supporta un'ampia gamma di back-end di archiviazione, quali file systems distribuiti, come il file systems di rete (**NFS**) e **GlusterFS**, gli storage persistente a livello di blocco, come **Amazon Elastic Block Store** (EBS) e Google Compute Engine, e infine container di storage come **Flocker**, un open-source Container Data Volume Manager per le applicazioni in Docker. L'aggiunta di un livello di archiviazione affidabile e disponibile a Kubernetes garantisce un'elevata disponibilità di workloads statici. Ogni componente del cluster di Kubernetes - etcd, API server, nodi - può essere configurato per garantire un'alta disponibilità. Le applicazioni hanno quindi il vantaggio di usufruire di un bilanciamento del carico di lavoro e di checks di controllo per garantire disponibilità.

3.3 Sicurezza

Essendo una piattaforma di orchestrazione, Kubernetes influisce su molte funzioni di sicurezza runtime. Questi includono autenticazione, autorizzazione, logging e isolamento delle risorse, oltre che più avanzati implicazioni come il posizionamento del carico di lavoro e la segmentazione della rete.

Però per affrontare correttamente la sicurezza, dobbiamo prima discutere i modelli di minaccia. Nell'ambiente di Kubernetes, ci sono generalmente quattro tipi di minacce che si applicano a prescindere dal modello di implementazione (con alcune eccezioni di casi limite)

Minaccia	Causa	Azioni
Attacchi esterni	API server, kubelet o componenti etcd sono compromessi	<ul style="list-style-type: none">• Esporre solo i servizi necessari di Kubernetes.• Applicare sempre l'autenticazione.• Configurare le giuste politiche di sicurezza della rete per tutti i servizi esposti.
Containers compromessi	Un container scala i privilegi per ottenere il controllo di altri container o del cluster	<ul style="list-style-type: none">• Utilizzare i meccanismi di isolamento di Kubernetes, come gli spazi dei nomi o la segmentazione della rete.• Utilizzare i controlli a livello di sistema operativo integrati.• Limitare il numero di container che possono essere eseguiti in una modalità privilegiata.
Credenziali compromesse	Un utente malintenzionato ottiene l'accesso al sistema	<ul style="list-style-type: none">• Imporre l'accesso ai privilegi minimi, il controllo degli accessi basato sui ruoli o altri controlli di accesso dettagliati.
Abuso di privilegi legittimi	Il sistema non è configurato correttamente e i privilegi dell'utente sono utilizzati in modo improprio.	<ul style="list-style-type: none">• Assicurarsi che tutti i componenti del sistema siano solidi.• Progettare e implementare correttamente l'autorizzazione.• Utilizzare l'architettura dei plugin di Kubernetes, che consente una logica di autorizzazione sofisticata.

Figura 3.2: Quattro tipi di minacce che si applicano indipendentemente dal metodo di deployment

La sicurezza in Kubernetes è configurata su più livelli. Gli endpoints dell'API sono protetti tramite Transport Layer Security (**TLS**), che garantisce l'autenticazione dell'utente utilizzando il meccanismo di autenticazione più sicuro disponibile. I cluster di Kubernetes hanno due categorie di utenti:

- Service account, gestiti direttamente da Kubernetes;
- Utenti normali, che si suppone essere gestiti da un servizio indipendente.

I service account gestiti da Kubernetes sono creati in automatico dall'API server. Ciascuna operazione che gestisce un processo in esecuzione all'interno di un cluster deve essere avviata da un utente autenticato; questo meccanismo garantisce la sicurezza del cluster.

Le applicazioni eseguite all'interno di un cluster Kubernetes possono beneficiare del concetto di *segreto* per un accesso sicuro ai dati. Un **segreto** è un oggetto Kubernetes che contiene una piccola quantità di dati sensibili, come password, token o chiave, che riduce il rischio di esposizione accidentale dei dati. Gli username e le password sono codificati in base64 prima di essere memorizzati all'interno del cluster Kubernetes. I pod possono accedere al segreto in fase di runtime tramite i volumi montati o le variabili d'ambiente.

Per permettere o limitare il traffico di rete ai pods, le politiche di rete possono essere applicate al deployment. Una policy di rete è una specifica di come i gruppi di pod possono comunicare tra loro e con altri endpoint di rete. Ciò può risultare utile in un deployment a più livelli per oscurare i pods in modo da non essere esposti ad altre applicazioni.

L'architettura di autenticazione e autorizzazione di Kubernetes è un processo astratto. Raffigurato nella Figura 3.3, questo processo richiama prima i moduli di autenticazione, poi i moduli di autorizzazione, e infine la richiesta viene passata attraverso i controlli di ammissione (se presenti), prima che raggiunga l'accesso a qualsiasi oggetto. Per abilitare l'autenticazione, l'amministratore del cluster deve configurare il server API per eseguire uno o più moduli di autenticazione al momento della creazione del cluster.

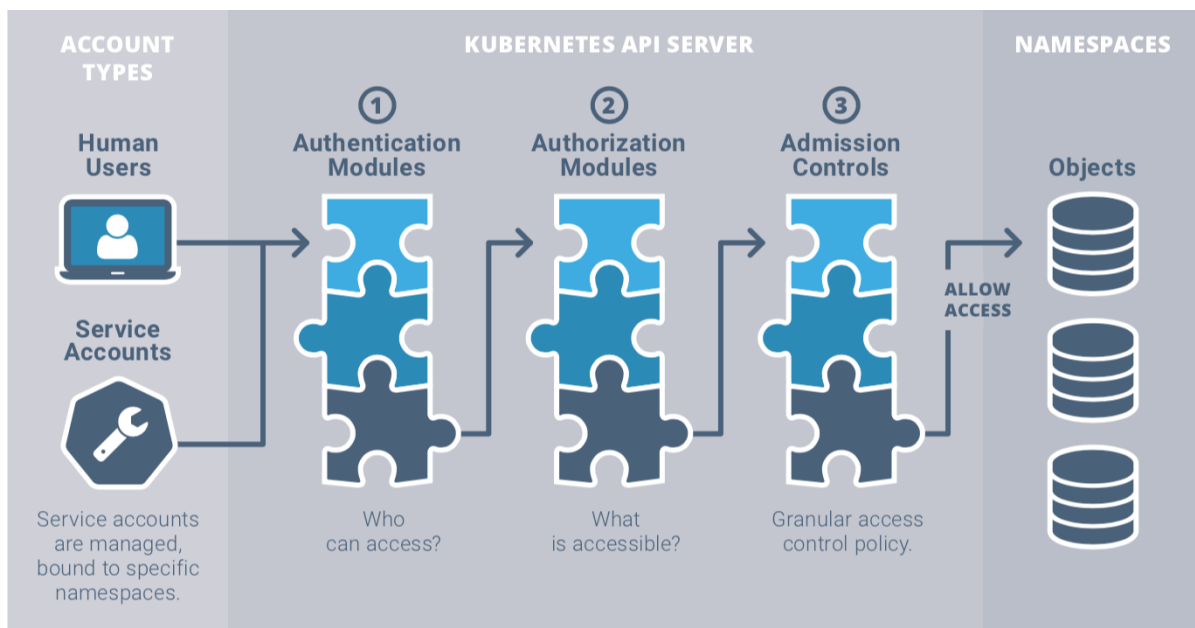


Figura 3.3: Utilizzo dei moduli di autenticazione, autorizzazione e Admission Control per controllare l'accesso API in Kubernetes.

L'autenticazione in Kubernetes è consentita con l'uso di certificati client, password, token statici, token di bootstrap e token Web JSON (JWT) per scopi di autenticazione. È possibile programmare una logica di autenticazione aggiuntiva utilizzando un proxy di autenticazione.

Per l'autorizzazione dell'utente, Kubernetes supporta più moduli di autorizzazione:

- **Node:** è un modulo di autorizzazione per scopi speciali che si occupa specificamente delle richieste API di kubelets. Un autore del nodo gestisce operazioni API di lettura, scrittura e autenticazione
- **ABAC** (Attribute Based Access Control): valuta gli attributi di una richiesta di accesso rispetto a un criterio di controllo degli accessi per concedere o negare l'accesso.
- **RBAC** (Role-Based Access Control): controlla l'accesso alle risorse API di Kubernetes in base al ruolo assegnato dell'utente.
- **Webhook:** Kubernetes utilizza Webhook per interrogare un servizio esterno sull'opportunità di concedere o negare la richiesta API corrente. Il meto-

do webhook fornisce a Kubernetes un mezzo per incorporare una politica di terze parti e potenzialmente implementare una logica arbitraria per le decisioni di autorizzazione.

3.4 Portabilità

Kubernetes è progettato per offrire una libertà di scelta rispetto alla scelta del sistema operativo da utilizzare, all'esecuzione dei container, all'architettura del processore, alle piattaforme cloud e al PaaS.

Infatti un cluster Kubernetes può essere configurato nelle principali distribuzioni Linux, quali CentOS, CoreOS, Debian, Fedora, Red Hat Linux e Ubuntu. Può essere eseguito in locale oppure in piattaforme cloud, come AWS, Azure e Google Cloud; su ambienti virtualizzati basati su KVM e vSphere. Gli utenti possono lanciare container in Docker o rkt runtimes, e nuovi container possono essere lanciati anche a tempo di esecuzione.

Inoltre attraverso la *federation* è anche possibile combinare cluster eseguiti tra più cloud provider differenti e in locale. Ciò porta le funzionalità del cloud ibrido ai workloads containerizzati. In questo modo i clienti possono facilmente spostare i carichi di lavoro da un target di deployment all'altro.

Capitolo 4

Struttura architetturale

Al giorno d'oggi le applicazioni containerizzate necessitano di un'infrastruttura sufficientemente solida per far fronte alle esigenze del clustering e allo stress dell'orchestrazione dinamica. Tale infrastruttura deve poter permettere lo scheduling, il monitoraggio, l'aggiornamento e il reallocaimento dei container tra i vari host. Deve essere in grado di trattare la computazione, l'archiviazione e le primitive di rete sottostanti come un pool di risorse.

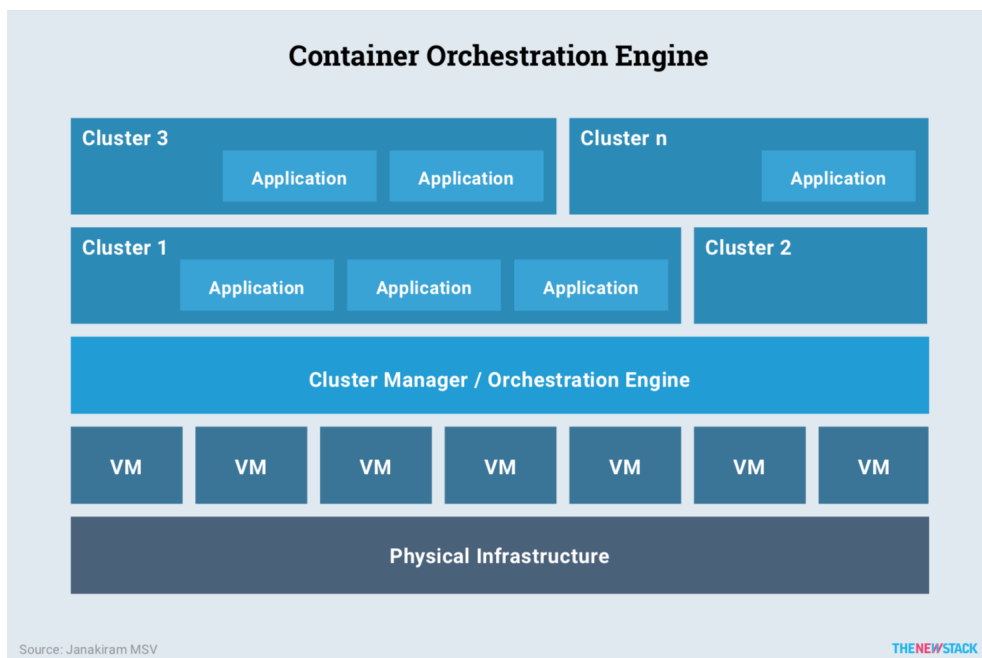


Figura 4.1: I livelli del sistema, da un punto di vista di orchestrazione dei container

Ogni carico di lavoro containerizzato dovrebbe essere in grado di sfruttare le risorse a esso esposte, inclusi CPU, unità di archiviazione e reti. Kubernetes è un gestore open source di cluster che astrae l'infrastruttura fisica sottostante, semplificando l'esecuzione di applicazioni containerizzate su larga scala. Un'applicazione, gestita per l'intero ciclo di vita da Kubernetes, è composta da un set di container raccolti e coordinati in una singola unità. Un efficiente livello di gestione del cluster permette a Kubernetes di poter disaccoppiare efficacemente l'applicazione dall'infrastruttura di supporto, come mostrato in figura 4.1. Una volta che l'infrastruttura di Kubernetes è stata configurata i team DevOps possono concentrarsi sulla gestione dei carichi di lavoro distribuiti piuttosto che gestire il pool di risorse sottostanti. L'API di Kubernetes può essere utilizzata per creare i componenti che fungono da blocchi fondamentali, o primitive, di microservizi. Questi componenti sono autonomi, il che significa che esistono indipendentemente dagli altri componenti.

4.1 Pod

Un pod è una collezione di uno o più container che funge da unità principale di Kubernetes per la gestione del carico di lavoro, fungendo da "confine logico" per i container che condividono contesto e risorse. Raggruppare container correlati in pod fa fronte alle sfide configurazionali introdotte quando la containerizzazione ha sostituito la virtualizzazione di prima generazione, rendendo possibile l'esecuzione di più processi dipendenti.

In fase di runtime, i pod possono essere ridimensionati creando set di repliche, che garantiscono che la distribuzione esegua sempre il numero desiderato di pod. Come già anticipato ogni pod è una raccolta di più container e per comunicare tra loro utilizzano le remote procedure calls (RPC), condividendo lo stack di archiviazione e di rete.

Negli scenari in cui i container devono essere accoppiati e co-localizzati, ad esempio un contenitore del server Web e un contenitore della cache, possono essere facilmente raccolti in un singolo pod. Un pod può essere scalato manualmente o tramite una feature chiamata Horizontal Pod Autoscaling (HPA).

Attraverso questo metodo il numero dei container all'interno di un pod viene aumentato proporzionalmente.

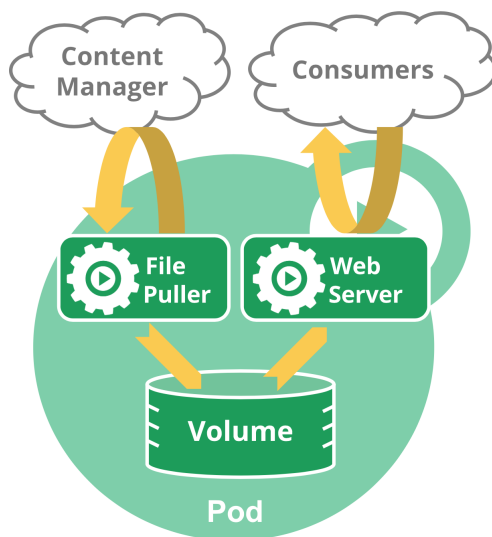


Figura 4.2: Un pod multi-container che contiene un file puller e un server Web che utilizza un volume persistente per l'archiviazione condivisa tra container

I Pods consentono inoltre una separazione funzionale tra development e deployment. Mentre gli sviluppatori si concentrano sul loro codice, gli operatori possono concentrarsi su una visione più ampia di come i container possono essere uniti insieme in un'unità funzionale. Il risultato ottenuto è una quantità ottimale di portabilità, dal momento in cui un pod non è altro che la rappresentazione di più immagini di container gestiti insieme.

4.2 Service (SOA)

Kubernetes, come la maggior parte dei sistemi distribuiti, è progettato secondo un'architettura orientata ai microservizi (**SOA**). In generale in questo tipo di architettura, ogni servizio ha un proprio contesto funzionale distinto e l'interazione con qualsiasi cosa al di fuori di tale contesto avviene tramite un'interfaccia astratta, in genere l'API di un altro servizio pubblico.

Nel caso di Kubernetes, un singolo pod o un ReplicaSet può essere esposto a client interni o esterni attraverso servizi, che associano un set di pod secondo un

criterio specifico. Quando viene creato un pod, viene assegnato un indirizzo IP accessibile solo all'interno del cluster. Ma non vi è alcuna garanzia che l'indirizzo IP del pod rimarrà invariato per tutto il suo ciclo di vita. Kubernetes può realllocare o re-istanziare i pod in fase di runtime, creando un nuovo indirizzo IP per il pod. Per compensare questa incertezza, i servizi assicurano che il traffico sia sempre indirizzato al pod appropriato all'interno del cluster, indipendentemente dal nodo su cui è pianificato. Ogni servizio espone un indirizzo IP e può anche esporre un endpoint DNS, entrambi i quali non cambieranno mai. I consumatori interni o esterni che necessitano di comunicare con un set di pod utilizzeranno l'indirizzo IP del servizio o il suo endpoint DNS più noto. In questo modo, il servizio funge da colla per il collegamento di pod con altri pod.

In Kubernetes qualsiasi oggetto API, incluso un nodo o un pod, può avere coppie chiave-valore ad esso associate o metadati aggiuntivi per identificare e raggruppare oggetti che condividono un attributo o una proprietà comune. Kubernetes si riferisce a queste coppie chiave-valore come etichette, in inglese "**labels**".

Un **selettore** è un tipo di criterio utilizzato per interrogare gli oggetti di Kubernetes che corrispondono a un valore di label. Questa potente tecnica consente di avere un basso accoppiamento tra oggetti. Inoltre possono essere creati nuovi oggetti che corrispondono al valore indicato dal selettore. Labels e selettori formano il meccanismo principale di Kubernetes per identificare i componenti ai quali si applica un'operazione.

Un **ReplicaSet** si basa su etichette e selettori per determinare quali pods saranno scalati. In fase di runtime i pod possono essere scalati attraverso i ReplicaSet garantendo così che ogni deployment abbia sempre il numero desiderato di pods. Ogni pod la cui etichetta corrisponde al selettore definito dal servizio verrà esposto il suo endpoint. Quando un'operazione di scaling viene avviata da un set di repliche, i nuovi pod creati da tale operazione inizieranno immediatamente a ricevere il traffico. Un servizio fornisce quindi il bilanciamento del carico di base instradando il traffico tra i pod di corrispondenza.

Questa architettura fornisce un meccanismo flessibile e liberamente accoppiato per l'individuazione dei servizi. Infatti la decostruzione di un sistema in un insieme di servizi complementari disaccoppia le operazioni tra le varie parti. Questa astrazione aiuta a stabilire chiare relazioni tra i servizi, il suo ambiente sottostante e i consumatori di quel servizio.

Creare chiare delineazioni può aiutare a isolare i problemi, ma permette inoltre di scalare ciascun servizio indipendentemente dagli altri. Questo tipo di progettazione orientata ai servizi è molto simile alla programmazione orientata agli oggetti.

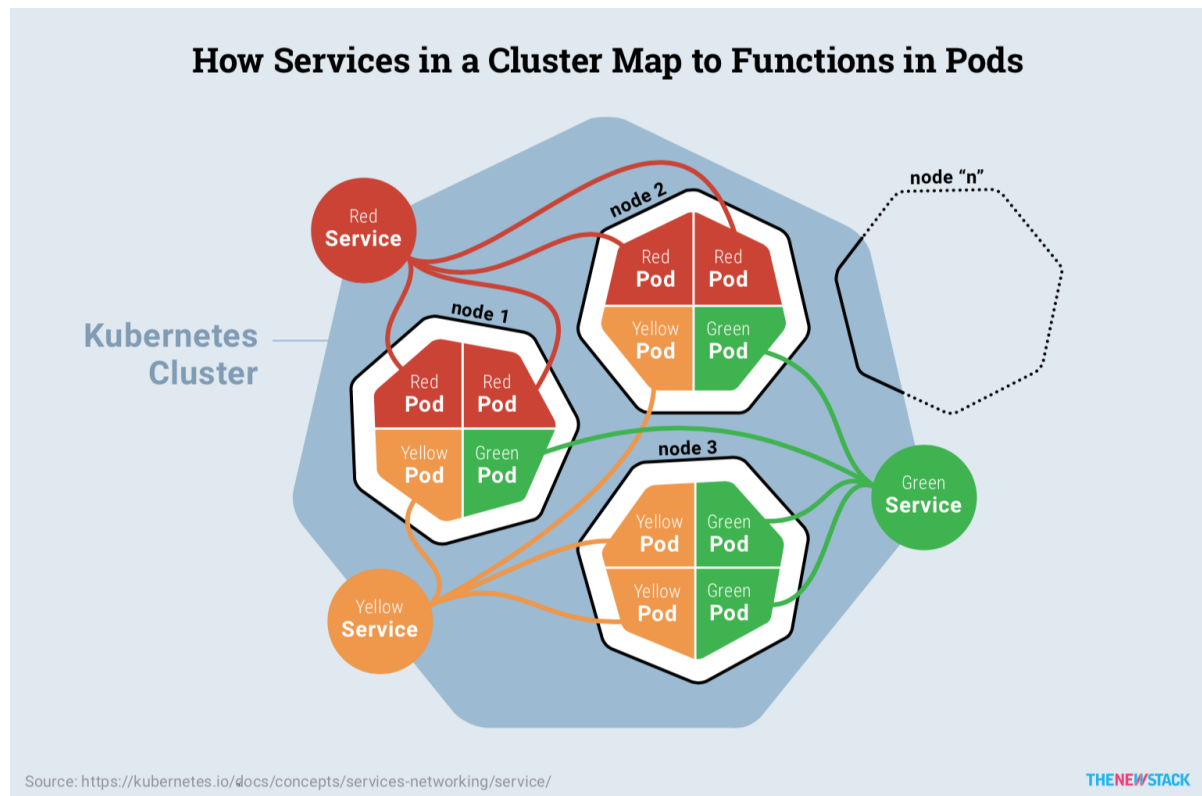


Figura 4.3: I servizi in Kubernetes

La figura 4.3 mostra la funzionalità dei servizi all'interno di un cluster Kubernetes. Sono presenti tre tipi di pod, distinti dai colori giallo, verde, rosso. Un replication controller ha scalato tali pod per eseguire istanze su tutti i nodi disponibili. Ogni classe di pod è esposto ai clients attraverso un servizio, rappresentato dal cerchio colorato.

Supponendo che ogni pod abbia una label della forma *color=value*, il suo servizio associato dovrebbe avere un selettore che lo abbinì.

Quando un client raggiunge il servizio rosso, la richiesta viene inoltrata a un qualsiasi pod che abbia la label *color=red*. Se viene aggiunto un nuovo pod in seguito

ad uno scaling, esso viene immediatamente rilevato dal servizio, in virtù della label e del selettore corrispondente.

In Kubernetes i servizi possono essere configurati per esporre pods a client interni o esterni. Un servizio esposto internamente è raggiungibile attraverso l'indirizzo **ClusterIP**, ovvero l'indirizzo del cluster. Questi tipi di servizio sono ad esempio i pods Database, che non necessitano di essere esposti esternamente. Invece quando un servizio necessita di essere raggiunto dall'esterno, è necessario esporre una porta specifica in ciascun nodo, chiamata **NodePort**.

Negli ambienti cloud pubblici, Kubernetes può eseguire il servizio di bilanciamento del carico configurato automaticamente per il routing del traffico ai suoi nodi.

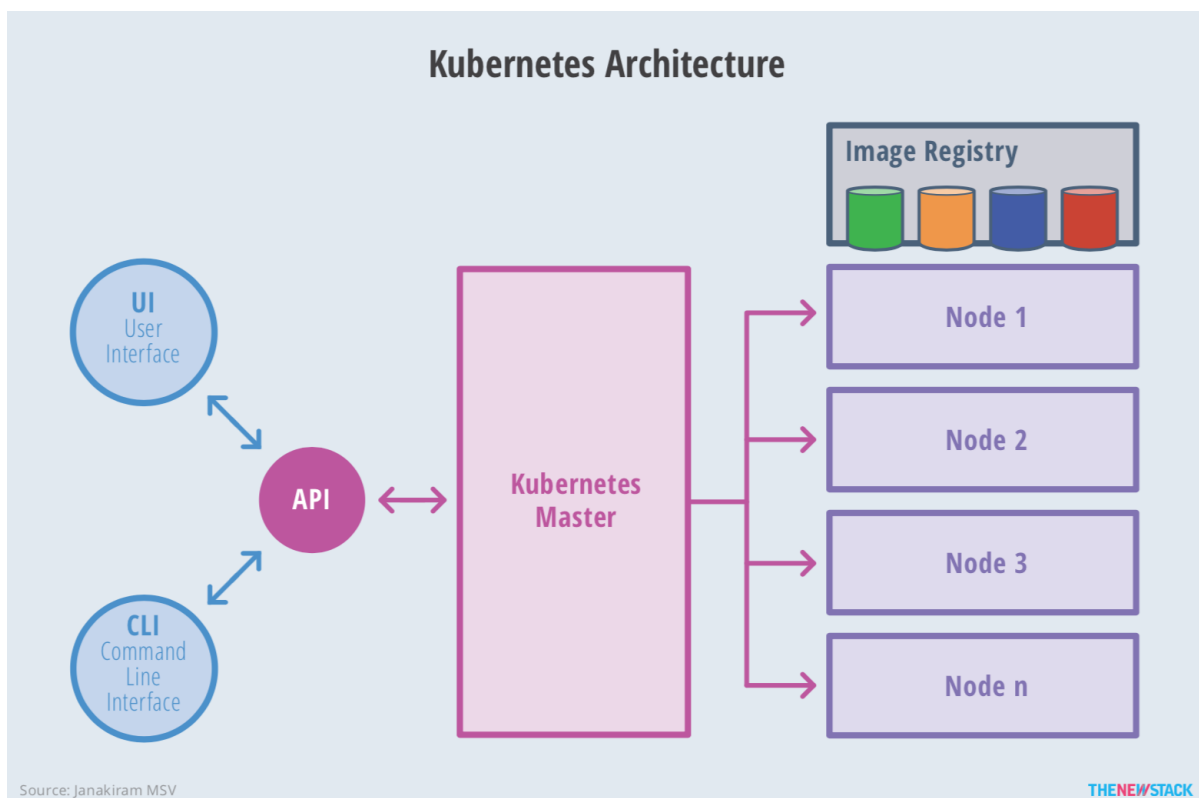


Figura 4.4: Componenti dell'architettura di Kubernetes

4.3 Nodo Master

Kubernetes, come la maggior parte delle piattaforme distribuite, è composto da almeno un nodo master e nodi di lavoro multipli. Il nodo **master** è responsabile per l'esposizione dell' API (application program interface), per la pianificazione del deployment e la gestione dell'intero cluster. Attraverso l'esposizione dell'API fornisce all'utente un'interfaccia per poter interagire con il sistema. Concettualmente, all'interno del nodo master non ci dovrebbero essere container in esecuzione. In alcuni casi, per garantire alta disponibilità, i nodi master possono essere anche più di uno.

La definizione degli oggetti in Kubernetes, così come i pods, i Replica sets e i servizi sono un compito del nodo master. Sono progettati per essere liberamente accoppiati, estensibili e adattabili a un'ampia varietà di carichi di lavoro. L'API fornisce questa estensibilità ai componenti interni, nonché estensioni e container in esecuzione su Kubernetes.

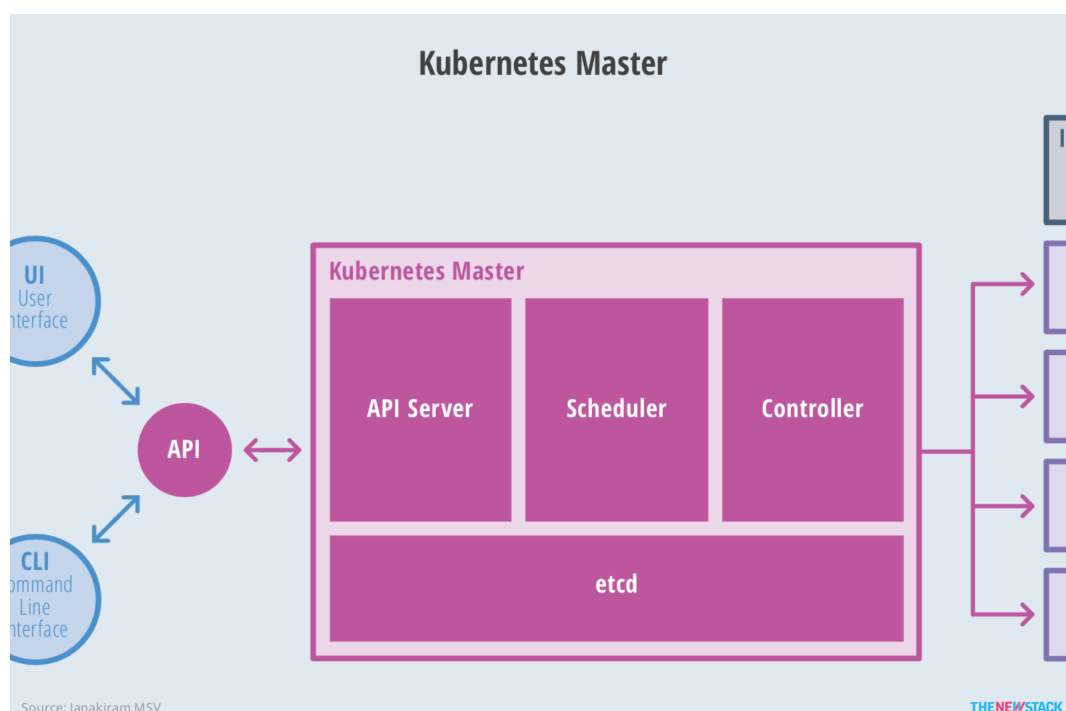


Figura 4.5: Componenti del nodo master

Come osservabile dalla figura 4.2, il nodo master si compone di più elementi, ciascuno spiegato nelle seguenti sottosezioni.

4.3.1 API Server

Il server API espone l'API di Kubernetes tramite JSON su HTTP, fornendo l'interfaccia REST per gli endpoint interni ed esterni dell'orchestratore per interagire con altri componenti del sistema. Impiega etcd-storage come componente di archiviazione persistente. La CLI, l'interfaccia utente web o un altro strumento possono inviare una richiesta al server API. Il server elabora e convalida la richiesta, quindi aggiorna lo stato degli oggetti API in etcd e infine istruisce il servizio appropriato per eseguirla. Ciò consente ai client di configurare carichi di lavoro e container tra i nodi worker.

4.3.2 etcd-storage

etcd è un database di CoreOS chiave-valore, distribuito e open-source, nonché persistente e leggero. Agisce come unica fonte per tutti i componenti del cluster di Kubernetes, infatti altri componenti e servizi guardano le modifiche a etcd per mantenere lo stato desiderato di un'applicazione. Il nodo master utilizza *etcd* per memorizzare i suoi stati, ad esempio i job pianificati, creati e distribuiti; dettagli e stato dei pod; informazioni sulle repliche, e altro.

4.3.3 Scheduler

Lo scheduler è responsabile di trovare un nodo corretto al quale assegnare un pod o un servizio, in base ad alcuni vincoli, ad esempio cerca di bilanciare l'utilizzo delle risorse tra i vari nodi e assicura che i pod si trovino su nodi che abbiano risorse libere a sufficienza, tenendo traccia dell'utilizzo delle risorse per garantire che il pod non superi l'allocazione consentita. Oltre a tenere traccia dei fabbisogni di risorse e della disponibilità delle risorse, possiede una varietà di altri vincoli e direttive forniti dall'utente; ad esempio, qualità del servizio (QoS), requisiti di affinità / anti-affinità e localizzazione dei dati. Un team operativo può definire il modello di risorse in modo dichiarativo. Lo scheduler interpreta queste dichiarazioni come istruzioni per l'approvvigionamento e l'allocazione del giusto insieme di risorse a ciascun carico di lavoro.

4.3.4 Controller

Il controller manager, o semplicemente controller, è un processo che al suo interno incapsula vari controllers. In kubernetes, il controller è identificato come un loop di controllo che periodicamente controlla lo stato del cluster attraverso *kube-apiserver* ed esegue le opportune modifiche per far sì che il cluster sia nello stato desiderato. Per stato desiderato s'intende il bilancio delle risorse dichiarate, utilizzate e richieste dal file di configurazione YAML dei pods, in base alle richieste e vincoli del sistema. Il controller mantiene lo stato stabile di nodi e pod, monitorando costantemente lo stato del cluster e i carichi di lavoro distribuiti su tale cluster. Ad esempio, quando un nodo diventa non sano, i pod in esecuzione su quel nodo potrebbero diventare inaccessibili. In tal caso, è compito del controller programmare lo stesso numero di nuovi pod in un nodo diverso. Questa attività garantisce che il cluster mantenga lo stato previsto in qualsiasi momento.

Il controller di Kubernetes svolge un ruolo cruciale nell'esecuzione dei carichi di lavoro containerizzati in produzione, rendendo possibile per un'organizzazione eseguire applicazioni containerizzate che vanno ben oltre i tipici scenari stateless e scale-out. Il controller manager supervisiona i controller di base di Kubernetes:

- **ReplicationController (ReplicaSet):** mantiene il numero di pod di una specifica distribuzione all'interno del cluster. Garantisce che un determinato numero di pod sia sempre in esecuzione in un qualsiasi nodo.
- **StatefulSet:** è simile al ReplicaSet, ma per i pod che necessitano persistenza e un identificatore ben definito.
- **DaemonSet:** assicura che uno o più container raccolti in pod sono in esecuzione su ciascun nodo del cluster. Questo è un controller speciale che forza l'esecuzione di un pod su ciascun nodo. Il numero di nodi eseguiti come *DemonSet* è direttamente proporzionale al numero di nodi.
- **job e cron job:** gestiscono l'elaborazione in background

Questi controller comunicano con il server API per creare, aggiornare ed eliminare le risorse che gestiscono, come i pod e gli endpoint del servizio.

Le applicazioni mission critical richiedono livelli più elevati di disponibilità delle

risorse. Attraverso l'uso di un set di repliche, Kubernetes garantisce che un numero predefinito di pod sia in esecuzione tutto il tempo. Ma questi pods sono apolidi ed effimeri. È molto difficile eseguire carichi di lavoro con stato, come un cluster di database o un grande stack di dati, per i seguenti motivi:

- A ciascun pod viene assegnato un nome arbitrario in fase di runtime.
- Un pod può essere pianificato su qualsiasi nodo disponibile a meno che non sia in vigore una regola di affinità, nel qual caso il pod può essere programmato solo su nodi che hanno un'etichetta o etichette specifiche specificate nella regola.
- Un pod può essere riavviato e reallocato in qualsiasi momento.
- Un pod non può mai essere referenziato direttamente con il suo nome o indirizzo IP.

A partire dalla versione 1.5, Kubernetes ha introdotto il concetto di **StatefulSets** per l'esecuzione di carichi di lavoro altamente disponibili. Un pod con stato che partecipa a un set stateful avrà i seguenti attributi:

- Un nome host stabile che verrà sempre risolto dal DNS.
- Un numero indice ordinale per rappresentare il posizionamento sequenziale del pod nel set di repliche.
- Una memoria stabile che è collegata al nome dell'host e al numero di indice ordinale.

Il nome host stabile con il numero di indice ordinale consente a un pod di comunicare con un altro in modo prevedibile e coerente. Questa è la differenza fondamentale tra un pod senza stato e un pod di stato.

4.4 Command Line Interface

La command line interface in kubernetes prende il nome di **kubectl** e permette di eseguire comandi all'interno del cluster Kubernetes interagendo con il nodo

master. Infatti comunica con kube-apiserver attraverso il servizio di API REST. Per eseguire un comando dal terminale è sufficiente inserire dopo kubectl il tipo di operazione desiderata (*create, get, describe, delete*), il type (ovvero il tipo di risorsa) e infine il nome della risorsa. Di seguito un esempio di comando: `"kubectl get pod pod1"`.

4.5 Node

Node è il cavallo di battaglia di Kubernetes, responsabile dell'esecuzione di workloads containerizzati. Il suo scopo è di esporre le risorse di elaborazione, di rete e di archiviazione alle applicazioni.

Un nodo può essere una macchina virtuale (VM) in esecuzione in un cloud o un server base metal all'interno del data center. I nodi workers possono essere molteplici all'interno di un cluster Kubernetes, e ciascuno di essi può avere un numero variabile di pod, mentre tutti possiedono tre principali componenti che permettono la corretta esecuzione dei container all'interno del nodo worker. Questi componenti sono *kubelet*, *kube-proxy* e *container runtime*.

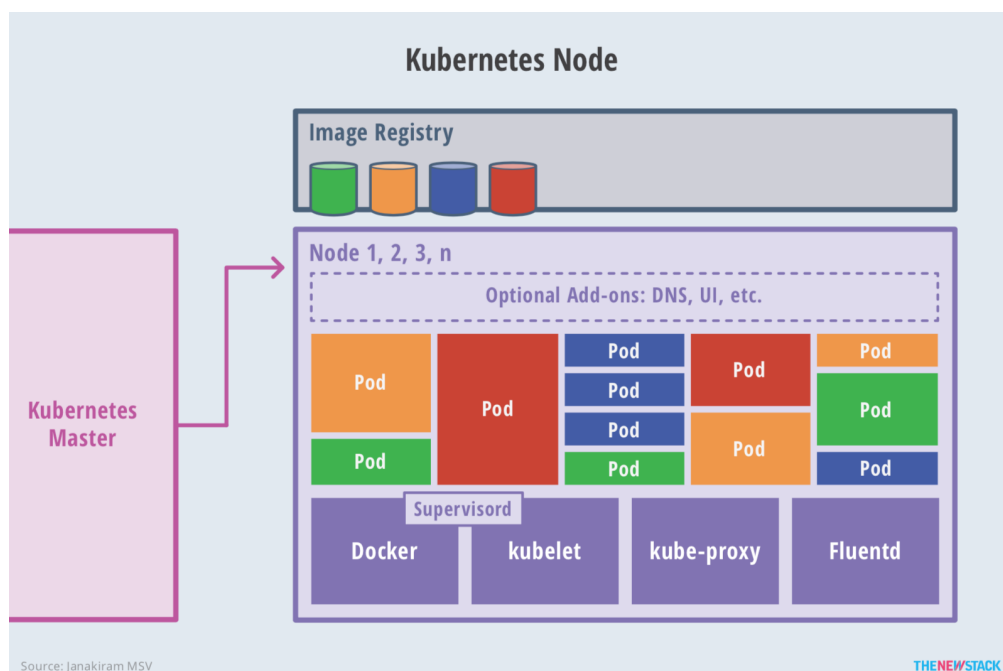


Figura 4.6: Componenti dei nodi workers

4.5.1 kubelet

Kubelet è il principale servizio di un nodo worker, responsabile della comunicazione con il nodo master. Garantisce che tutti i container su un nodo siano in buona salute, inoltre interagisce con il servizio di container runtime per eseguire operazioni quali l'avvio, l'arresto, e la manutenzione di containers. Kubelet ottiene la descrizione della configurazione di un pod dal kube-apiserver tramite API REST, per poi ordinare al servizio di container runtime di eseguire i container appropriati. Questo componente riporta anche al master lo stato di salute dell'host su cui è in esecuzione. Lo stato del nodo viene trasmesso al master ogni pochi secondi tramite messaggi heartbeat. Se il master rileva un errore del nodo, il controller di replica osserva questo cambiamento di stato e pianifica i pod su altri nodi sani.

4.5.2 kube-proxy

Kube-proxy è implementato come proxy di rete e bilanciamento del carico. Instrada il traffico al container appropriato in base al suo indirizzo IP e al numero di porta di una richiesta in arrivo. Esegue l'inoltro delle richieste ai relativi pod/containers tra le varie reti isolate all'interno del cluster.

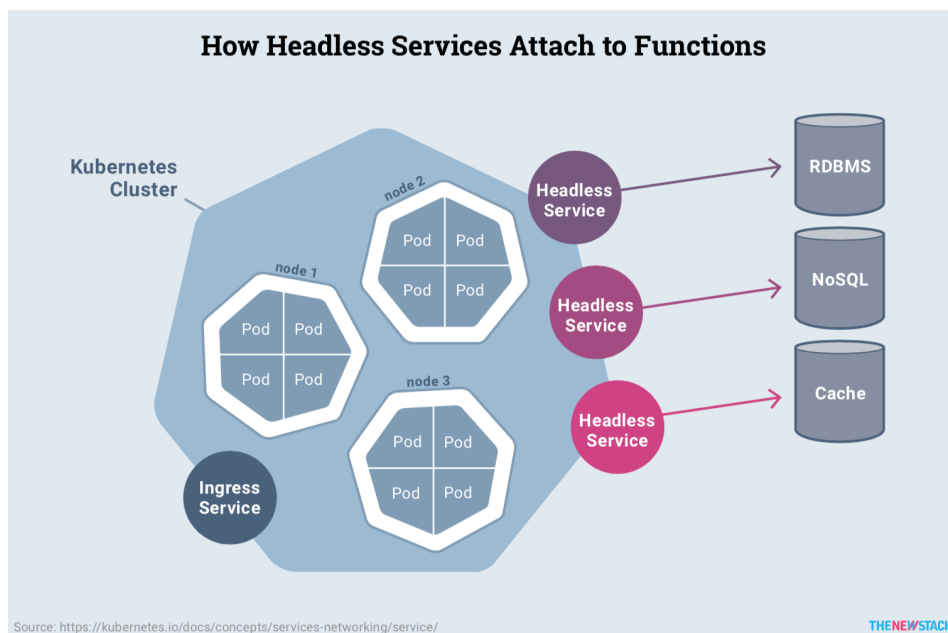


Figura 4.7: Servizi headless con punti di connessione tra database e servizi di stato

Come illustrato nella figura 5.4 è possibile utilizzare il **servizio headless** per dirigere i pods a servizi esterni, come cache, oggetti di archiviazione e database. Ciò funziona essenzialmente come un servizio ma senza la necessità di un kube-proxy o di un bilanciamento del carico. Una chiamata ad un servizio headless determina l'indirizzo IP del cluster che il servizio seleziona in modo specifico. In questo modo, è possibile utilizzare una logica personalizzata per selezionare l'indirizzo IP, bypassando la normale routine.

4.5.3 Container runtime

Container runtime è il software responsabile dell'esecuzione dei container all'interno del nodo. Dopo che un pod è schedulato sul nodo, il runtime estrae le immagini specificate dal pod dal registro. Quando un pod termina, il runtime uccide i container che appartengono al pod. Kubernetes può comunicare con qualsiasi runtime container compatibile con OCI (Open Container Initiative), inclusi Docker e rkt.

4.5.4 Logging Layer

L'orchestratore fa frequentemente uso del logging come mezzo per raccogliere l'utilizzo delle risorse e le metriche delle prestazioni per i container su ciascun nodo, come CPU, memoria, file e utilizzo della rete. Cloud Native Computing Foundation produce quello che definisce un livello di registrazione unificato da utilizzare con Kubernetes o altri orchestratori, chiamato **Fluentd**. Questo componente produce metriche che il controller master di Kubernetes deve tenere traccia delle risorse cluster disponibili e dello stato dell'infrastruttura generale.

Capitolo 5

Deployment patterns

Come introdotto nel Capitolo 2, la rapida crescita di Kubernetes nell'ecosistema dei container ha portato a molteplici modelli di deployment, che vanno dal fai-da-te alle forme di cluster completamente automatizzate e gestite. Indipendentemente dal modo in cui viene distribuito, gli sviluppatori e i team di sviluppo seguono un flusso di lavoro standardizzato e coerente per la gestione del ciclo di vita delle applicazioni containerizzate. Questo è uno dei vantaggi chiave di Kubernetes. I clienti che utilizzano Kubernetes hanno accesso a un ampio spettro di modelli di deployment disponibili, ognuno con i suoi vantaggi e svantaggi:

- **Deployment personalizzato:** una distribuzione personalizzata offre la massima scelta ai clienti tra un'ampia gamma di ambienti di distribuzione target, configurazioni della macchina, sistemi operativi, back-end di archiviazione, plug-in di rete e configurazioni ad alta disponibilità. Pur offrendo scelta e controllo, la responsabilità di mantenere i cluster spetta esclusivamente al cliente. Nelle distribuzioni personalizzate, i clienti possiedono l'intero stack che alimenta il cluster di produzione. Dalle risorse di calcolo, di rete e di archiviazione sottostanti al registro delle immagini, l'utente è responsabile dell'installazione, della configurazione e della gestione dell'intero stack. Nel caso del cloud pubblico, alcune risorse, come macchine virtuali (VM) e dispositivi di archiviazione a blocchi, sono gestite dal provider IaaS.
- **Managed Kubernetes Cluster:** le piattaforme gestite di Kubernetes offrono il meglio di entrambi i mondi: la scelta dell'infrastruttura combinata con

cluster privi di manutenzione. Le organizzazioni che non dispongono delle competenze necessarie per configurare, configurare e gestire distribuzioni su larga scala possono optare per piattaforme gestite di Kubernetes. Poiché i fornitori di piattaforme gestiscono da remoto i cluster, gli utenti possono concentrarsi sullo sviluppo dell'applicazione invece di mantenere l'infrastruttura del container. Rispetto alle distribuzioni self-hosted, le offerte gestite di Kubernetes sono più costose, in quanto vanno aggiunti i costi dell'infrastruttura e di gestione del cluster. Ma il vantaggio di aggiornamenti periodici, patching, servizi di sicurezza e monitoraggio delle piattaforme gestite di Kubernetes può compensare il costo a lungo termine.

- **Container as a Service:** se confrontato con altri modelli di implementazione, CaaS fornisce il percorso più breve per Kubernetes. I clienti possono avere un cluster completamente configurato, altamente disponibile e sicuro in pochi minuti. CaaS è dotato di funzionalità come monitoraggio integrato, registrazione, auto-scaling, auto-aggiornamento e auto-health. In questo modo i team DevOps non devono occuparsi della gestione dei cluster, con risparmi a lungo termine sulla gestione dell'infrastruttura. Uno svantaggio è la mancanza di controllo. I provider di cloud potrebbero impiegare più tempo per aggiornare le versioni di Kubernetes con la probabilità di non supportare tutti i componenti aggiuntivi e le funzionalità. I clienti non possono scegliere il back-end di archiviazione.
- **Platform as a Service:** mentre altri modelli sono orientati alle operazioni, PaaS è orientato agli sviluppatori e viene preferita da un'organizzazione che sta prendendo in considerazione un livello uniforme e coerente che nasconde la complessità dell'orchestrazione dei container e gestione dell'infrastruttura. Scalabilità, monitoraggio e logging sono integrati nella piattaforma, che possono essere configurati tramite API o console basate sul web. Sebbene PaaS riduca la complessità, manca di flessibilità. La mancanza di personalizzazione è uno dei limiti di queste piattaforme. Se confrontato con altri modelli di delivery, il modello di licenza PaaS lo rende leggermente costoso per gli utenti.










































Features	Self Hosted, Custom deployment	Managed Kubernetes Cluster	Container as a Service (CaaS)	Platform as a Service (PaaS)
Ability to customize	 High	 Medium/High	 Medium/Low	 Low
Overall costs (SW e Infrastructure)	 High	 Medium	 High	 High
Staffing and support costs	 High	 Medium	 Low	 Low
Admin skills	 High	 Medium	 Low	 Low
Portability around cloud	 Low	 High	 Medium (varies by providers)	 Medium (varies by providers)
Container image registry	 Not included	  Varies by provider	 Included	 Included
Security and monitoring services	 Not included	 Included	 Included	 Included
High-availability features	 Not included	 Not included	 Included	 Included
Built-in infrastructure auto- scaling	 Not included	 Not included	 Included	 Included
Complete app lifecycle management	 Not included	 Not included	 Not included	 Included

Figura 5.1: Confronto tra i diversi livelli di controllo, costi e funzionalità tra i vari deployment patterns

Capitolo 6

Aspetti analitici e criticità

6.1 Code Quality

La qualità del codice è stata analizzata attraverso Codebeat, uno strumento che valuta automaticamente la qualità del codice. Questo tool controlla la qualità del codice calcolando un insieme di metriche correlate alla qualità del software, all'estensibilità e alla manutenibilità. In particolare sono state valutate la complessità totale e la duplicazione di codice.

Dal punto di vista della **duplicazione di codice**, il numero di duplicati nel codice è considerato una delle metriche più importanti quando si misura la qualità del codice in quanto può causare difficoltà durante la manutenzione, poiché cambiare un'istanza di codice richiederebbe la replica anche in altri casi. Dall'analisi di codebeat, in un particolare file chiamato *api.pb.go* lo sviluppatore ha usato le stesse linee di codice in diverse funzioni; ciò significa che devono cambiare tutto il codice nelle funzioni corrispondenti ogni volta che devono modificare la logica. Per quanto riguarda la **complessità del codice**, l'elevato numero di complessità indica che il codice contiene troppa logica e probabilmente dovrebbe essere scomposto in elementi più piccoli. La complessità del codice può essere misurata utilizzando diverse metriche come il numero di argomenti e valori di ritorno in una funzione; e il numero di casi condizionali. Uno degli esempi di questo può essere visto nell'immagine seguente.

In questo esempio, ci sono otto argomenti nella funzione *AttachContainer* che potrebbe causare confusione quando si usa questa funzione.

The screenshot shows a Go code editor with a function `AttachContainer` in `kubelet.Kubelet`. The function signature is `func (kl *Kubelet) AttachContainer(podFullName string, podUID types.UID, containerName string, stdin streamingRuntime, ok := kl.containerRuntime.(kubecontainer.DirectStreamingRuntime))`. The error message at the top says "Too many function arguments kubelet.Kubelet.AttachContainer" with a "critical" icon and "0 arguments" listed. The code snippet shows lines 1653 to 1667, including a `if` statement for `lok`, a `return` statement for an error, and a `return` statement for the `AttachContainer` call.

Figura 6.1: Esempio di complessità del codice

6.2 Utility tree

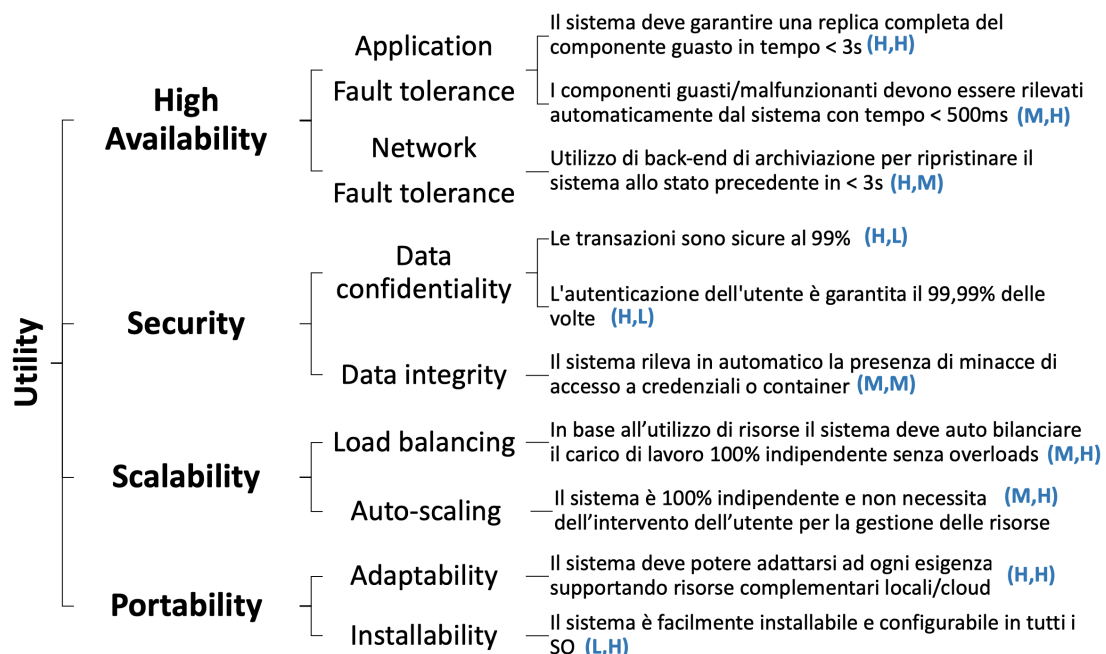


Figura 6.2: Utility tree di Kubernetes con priorità degli scenari

Nella figura 6.2 è possibile osservare l'**utility tree** relativo a Kubernetes. In generale, l'*utility tree*, come dice appunto la parola *utility*, indica la bontà complessiva del sistema. Gli attributi di qualità costituiscono il secondo livello dell'albero essendo componenti di utilità, e corrispondono agli stessi drivers architetturali discussi nel capitolo 2.

Parliamo ora in dettaglio di questo livello e dei livelli seguenti:

- **Availability:** Questo driver può essere raggiunto grazie a
 - *Fault tolerance*: una proprietà molto importante che indica la tolleranza ai guasti, ovvero la capacità del sistema di supportare un'alta disponibilità di applicazioni e infrastrutture. Sia la *fault tolerance* rispetto alle applicazioni che alle infrastrutture possiede un'alta priorità in quanto si vuole garantire la massima efficienza.
- **Security:** Questo driver può essere raggiunto grazie a
 - *Data confidentiality*: assicura che le transizioni siano sicure e l'autenticazione è sempre garantita. Questo task ha un'alta priorità, che per Kubernetes non è difficile da soddisfare in quanto adotta un'autenticazione a moduli molto sicuro.
 - *Data integrity*: assicura che i dati e i contenuti relativi all'utente siano integri, proteggendoli da eventuali minacce. Questo task ha una priorità media e lo scenario è più difficile da soddisfare in quanto è necessario un utilizzo consapevole da parte dell'utente.
- **Scalability:** Questo driver può essere raggiunto grazie a
 - *Load balancing*: il bilanciamento del carico è una qualità molto importante per Kubernetes, che viene soddisfatta grazie all'auto-scaling, che prende in considerazione la quantità d'utilizzo delle risorse del sistema, garantendo così che il sistema sia sempre bilanciato.
- **Portability:** Questo driver può essere raggiunto grazie a
 - *Adaptability*: la capacità del sistema di adattarsi a nuove risorse disponibili sia sulla propria macchina o in cloud. In Kubernetes questa qualità

ha un'alta priorità ed è in costante sviluppo grazie a *federation* con cui è possibile combinare più cluster in un deployment ibrido.

- *Installability*: qualità del software di essere facilmente configurabile ed installabile. Purtroppo Kubernetes supporta solo sistemi Linux ma non è ancora integrabile con Windows.

Capitolo 7

Stili architetturali simili: Docker Swarm

Tra i tool concorrenti a Kubernetes che hanno un'architettura simile ed offrono un servizio di orchestrazione di container abbiamo **Docker Swarm** e **Marathon**. Marathon è una piattaforma di orchestrazione di applicazioni e servizi per Apache Mesos. Docker Swarm invece, che è integrato in Docker come *Swarm Mode*, è un orchestratore per i container Docker. Docker Swarm e Kubernetes hanno scolpito e cementato le loro posizioni nell'ecosistema dei container, ed entrambi questi strumenti consentono di gestire un cluster di server su cui sono in esecuzione uno o più servizi, ma sebbene entrambe le piattaforme di orchestrazione open source forniscano molte delle stesse funzionalità, ci sono alcune differenze fondamentali tra il modo in cui queste due operano. Kubernetes supporta richieste più elevate con maggiore complessità, mentre Docker Swarm offre una soluzione semplice con cui è facile iniziare. Docker Swarm è stato molto popolare tra gli sviluppatori che preferiscono implementazioni rapide e semplicità. Kubernetes è un tool molto più efficiente, in quanto offre una maggiore scalabilità e automazione; inoltre possiede delle proprie librerie per il monitoraggio e i logging, e ciò viene a mancare in Docker Swarm, che deve ricorrere ad applicazioni di terze parti per sopperire a queste features. In Kubernetes se un nodo fallisce, esso viene rimpiazzato così velocemente da non accorgersi, a meno che non si esegua una dettagliata risoluzione dei problemi. Nella seguente tabella viene approfondita la "rivalità" tra Kubernetes e Docker Swarm.

	Kubernetes	Docker Swarm
Scalability	Supporta cluster con un massimo di 100 nodi, l'99% di tutte le chiamate API viene restituito in meno di 1 secondo e il 99% dei pod e i relativi container hanno un tempo d'avvio entro 5 secondi.	Testato per supportare 1000 nodi e 30.000 container senza alcuna differenza evidente tra il tempo di avvio del 1° o 30.000° nodo.
Availability	Altamente disponibile. I pod sono distribuiti tra i nodi worker. I nodi guasti vengono rilevati automaticamente e gestiti per evitare tempi di inattività.	Altamente disponibile. I container sono distribuiti tra i nodi Swarm per ridondanza e alta disponibilità. Lo Swarm Manager gestisce le risorse su larga scala e viene replicato per garantire che siano altamente disponibili.
Load-balancing	Kubernetes consente ai pod di essere definiti come un servizio. È quindi possibile impostare un bilanciamento del carico per accedere a questi servizi.	Lo swarm manager utilizza un bilanciamento del carico in ingresso per esporre i servizi che vuole rendere visibili esternamente allo swarm. Inoltre utilizza anche il load-balancing internamente.
Auto-scaling	Eccellente. Target come numero di pod e CPU-utilizzo-per-pod sono disponibili direttamente tramite l'API ed è sufficiente impostare questi parametri e K8 assegna automaticamente le risorse per mantenerli.	Non supporta l'auto-scaling, ma è possibile dichiarare il numero di attività che si desidera eseguire e Swarm Manager si adatterà per mantenere questo stato quando si scala in su o in giù.
Performance	Kubernetes ha un'architettura più complessa e flessibile di Docker e offre maggiori garanzie. Purtroppo ancora queste cose rallentano le prestazioni.	Docker ha un'architettura più semplice ed in termini di prestazioni è più veloce. Secondo alcuni test è 5X più veloce di K8 in termini di start di un cluster di container e scalabilità.
Easy to use	Kubernetes è più difficile da configurare e da utilizzare in quanto più complesso.	L'API di Docker Swarm condivide la CLI di Docker, inoltre è più indicato per implementazioni rapide e semplici

Bibliografia

- [1] Alex Williams, *The State of the Kubernetes Ecosystem*, TheNewStack, 2017.
- [2] Alex Williams, *Kubernetes Deployment e Security Patterns*, TheNewStack, 2018.
- [3] Alex Williams, Benjamin Ball, Gabriel Hoang Dinh, Lawrence Hecht, *Use Cases for Kubernetes*, TheNewStack.
- [4] Arsh Modak, Chaudhary, Paygude, Idate, *Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes?*, 2018 2nd International Conference, IEEE.
- [5] Enreina Annisa Rizkiasri, Francisco Morales, Haris Suwignyo, Mohammad Riftadi, *Kubernetes - Container Orchestration*, Delft Students on Software Architecture, 2018.
- [6] Kate Matsudaira, *Scalable Web Architecture and Distributed Systems*, The architecture of Open Source Applications, vol 2.