ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Kubernetes su OpenStack: deployment automatizzato su un cluster ARM di un private cloud per l'orchestrazione di container

Tesi di laurea in
SYSTEMS INTEGRATION

Relatore
Prof. Vittorio Ghini

Candidato
Ruben Ceroni

Terza Sessione di Laurea
Anno Accademico 2020-2021

Sommario

Kubernetes è ormai lo standard de facto dell'industria per l'orchestrazione di container. Tipicamente viene costruito su infrastruttura fornita da cloud provider pubblici, astraendo all'utente tutti gli aspetti di basso livello necessari alla messa in essere del cluster.

L'obiettivo di questa tesi è di riprodurre in scala ridotta un'infrastruttura cloud privata su hardware a basso costo ed utilizzarla per costruire un cluster Kubernetes.

In questo modo è possibile analizzare a fondo tutti gli aspetti inerenti alla costruzione di un cloud privato ed al suo utilizzo tramite Kubernetes.

Per raggiungere gli obiettivi preposti è stato necessario, in seguito all'identificazione dell'hardware, costruire un bare metal cloud, gestito attraverso MAAS. Sfruttando questo primo livello è poì stato possibile installare il cloud OpenStack, utilizzando Kolla-Ansible. Su questa base è stato possbile installare Kubernetes, includendo meccanismi di cluster autoscaling.

Il tutto è stato effettuato automatizzando ove possibile i processi di istanziazione e configurazione delle risorse tramite strumenti di IaC: Terraform e Ansible.

I risultati ottenuti hanno dimostrato la fattibilità degli obiettivi preposti, ottenendo un sistema con performance e caratteristiche adeguate, fornendo allo stesso tempo un'introspettiva del processo di gestione di un cloud privato.

Ringraziamenti

Con la conclusione di questo percorso di studi desidero ringraziare tutti i professori del corso di Laurea triennale e magistrale di Ingegneria e Scienze Informatiche a Cesena, che hanno sempre saputo stimolare la mia voglia di apprendimento e l'interesse per le materie.

In particolare ringrazio il Prof. Vittorio Ghini che, come mio relatore, mi ha accompagnato e sostenuto durante questo lavoro, rimanendo sempre disponibile per supporto, consigli e guida.

Ringrazio la mia famiglia e gli amici, che sono sempre stati al mio fianco e mi hanno supportato nei momenti di difficoltà.

Una menzione particolare va a tutti gli ormai ex-coinquilini di Bocca, grazie ai quali ho trovato a Cesena un ambiente quasi familiare che mai mi sarei immaginato.

Ringrazio anche Perla, che in quest'ultimo periodo mi ha incoraggiato e dato la forza di completare questo percorso.

Ringrazio inoltre il progetto Trashware Cesena e tutti i suoi collaboratori presenti e passati, che oltre a fornirmi parte del materiale necessario, mi ha dato occasione di svagarmi ed applicare le mie conoscenze in maniera pratica. Ringrazio anche l'Associazione S.P.R.I.Te., che ha reso questi anni a Cesena molto più divertenti ed interessanti, dandomi anche l'occasione di relazionarmi con persone e mettermi in gioco.

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Introduzione

Al giorno d'oggi l'architettura delle applicazioni sviluppate si sta evolvendo, allontanandosi dal tradizionale monolite, andando verso un'architettura a microservizi.

La gestione di queste applicazioni, suddivise in svariati container, risulta sempre più complessa, richiedendo appositi tool che ne gestiscano l'orchestrazione.

Tra di questi spicca Kubernetes, che negli ultimi anni è diventato lo standard de facto per l'orchestrazione di container. Esso viene tipicamente hostato su piattaforma cloud pubbliche, come Azure e AWS, che si occupano di tutti gli aspetti di gestione.

Per analizzare a fondo questa infrastruttura si è deciso di riprodurre in scala ridotta un cluster, utilizzando hardware a basso costo per facilitare l'accessibilità a chiunque sia interessato. Questo verrà fatto costruendo un cloud privato utilizzando OpenStack. L'infrastruttura messa a disposizione dal cloud potrà poi essere utilizzata per creare un cluster Kubernetes, sul quale verrà implementato un meccanismo di scalabilità automatica.

Mediante la costruzione di questo cluster sarà quindi possibile analizzare a fondo aspetti di architettura che tipicamente non escono dal datacenter e rimangono nascosti all'utilizzatore di cloud pubblici e container orchestration engine.

Un aspetto a cui verrà posta particolare attenzione sarà l'automazione dell'intero processo di messa in essere del cluster, che include l'installazione del sistema operativo, l'installazione di OpenStack e l'installazione di Kubernetes. Attraverso strumenti appositi si renderà quindi automatica e riproducibile la costruzione dell'ambiente finale.

Questo documento di tesi sarà strutturato nel seguente modo:

Strumenti Descrizione ed introduzione ai principali strumenti e le tecnologie utilizzate in questa tesi.

Obiettivi Definizione degli obiettivi da raggiungere.

Testing preliminare Descrizione delle prove preliminari effettuate e degli effetti che hanno avuto sulla progettazione.

Progettazione del cluster Descrizione dell'architettura del sistema e di tutti gli aspetti di progettazione degli applicativi.

Costruzione del cluster Descrizione degli aspetti relativi alla messa in essere del cluster e degli eventuali problemi riscontati e le soluzioni adottate.

Validazione dei risultati Valutazione delle performance e delle capabilità del cluster costruito, effettuando prove funzionali.

Conclusioni e svliuppi futuri Riassunto dei contributi principali e descrizione dei possibili sviluppi futuri.

Capitolo 2

Strumenti

In questo capitolo verranno descritti i principali strumenti e le tecnologie utilizzate nel lavoro di tesi, le possibili alternative ed il loro contesto di utilizzo.

2.1 Cloud Computing

Il cloud computing consiste nella disponibilità di risorse di computazione configurabili, in modalità on-demand attraverso Internet.
È definibile attraverso le sue caratteristiche essenziali [11]:

On-demand self service Un utente può creare risorse in maniera indipendente, senza richiedere l'intervento di un operatore;

Accessibilità via rete I servizi offerti dal cloud sono raggiungibili attraverso la rete, indipendentemente dal dispositivo usato per accedervi;

Resource pooling Le risorse di computazione offerte dal provider sono aggregate ed assegnate dinamicamente agli utenti in base alla necessità. L'utente non è a conoscenza dell'esatta posizione della risorsa utilizzata;

Elasticità rapida Le capacità di computazione possono essere espanse o ristrette in base alla domanda, in maniera automatica. La visione dell'utente sarà di una capacità di computazione infinita;

Servizio misurato Il consumo di risorse viene tracciato per consentire ottimizzazione e controllo su di esse.

2.1.1 Tipologie di cloud

Sono stati definiti alcuni modelli di deployment per infrastrutture cloud, descritti di seguito.

Public cloud

Un cloud pubblico è un insieme di risorse virtualizzate, gestite da una terza parte, che le mette a disposizione degli utenti attraverso un'interfaccia self-service. Tipicamente vengono applicati meccanismi di billing basati sul tempo di utilizzo delle risorse. Alcuni esempi di public cloud sono Amazon Web Services, Microsoft Azure, Google Cloud.

Private cloud

Un cloud privato è un cloud interamente dedicato ad un utente o una singola organizzazione. Viene tipicamente eseguito on-premise o in uno spazio dedicato all'interno di un datacenter.

Un private cloud è la soluzione adottata da aziende o enti che, per motivi di sicurezza o legislativi non possono appoggiarsi su un cloud pubblico, ma vogliono beneficiare delle risorse on-demand messe a disposizione da esso. La scelta può dipendere anche dal tipo di workload che si vuole fare eseguire al cloud: mentre i cloud pubblici sono più orientati a carichi di lavoro *stateful*, invece i carichi di lavoro *stateless*, come la virtualizzazione di funzione di rete, sono meglio supportati da un cloud privato [18].

Hybrid cloud

Un *hybrid cloud* è la composizione di un ambiente cloud con un secondo ambiente di computazione che estende il primo, tipicamente si intende l'estensione di un cloud pubblico con un cloud privato. Tipicamente questo approccio è dettato o da esigenze di maggiore sicurezza per certe applicazioni o dalla necessità di eseguire carichi di lavoro particolari.

La connessione fra i due cloud è realizzata attraverso VPN o connessioni WAN, che permettono l'interazione fra i cloud, sfruttando middelware appositi [16].

2.1.2 Tipologie di servizi

Una piattaforma cloud fornisce all'utente servizi a vari livelli di astrazione, di seguito vengono descritti i principali modelli di servizio.

Infrastructure as a Service

Il modello *IaaS* offre all'utente la possibilità di creare le risorse base necessarie per la computazione tra cui processing, storage e rete. In questo modo è possibile creare una infrastruttura completamente virtualizzata per il deployment di software arbitrario.

Platform as a Service

Un servizio *PaaS* mette a disposizione un ambiente per il deployment di applicazioni create dall'utente su infrastruttura gestita dal cloud provider.

Software as a Service

SaaS consiste nell'utilizzare applicazioni sviluppate dal cloud provider e messe a disposizione attraverso la sua infrastruttura, tipicamente attraverso un browser web;

Function as a Service

Chiamato anche *serverless*, *FaaS* mette a disposizione un ambiente per il deploy di applicazioni altamente scalabile, basato su eventi. Il cloud provider, al verificarsi di un determinato trigger, istanzierà una macchina virtuale effimera dedicata alla singola esecuzione dell'applicazione, per poi deallocarla al termine dell'esecuzione.

2.1.3 Cluster computing

Un cluster di computer è un set di computer che operano collaborando fra di loro, visibili dall'esterno come un singolo sistema. Sono connessi fra loro attraverso una rete LAN ad alta velocità ed eseguono applicazioni in maniera distribuita.

Tipicamente alle macchine è assegnato un ruolo, in base al servizio che offrono al cluster: alcune macchine, ad elevato numero di core e con grandi quantità di RAM si occupano di computazione, mentre macchine con un elevato numero di dischi offrono servizi di storage dati e macchine con svariate GPU offrono servizi di accellerazione per applicazioni di intelligenza artificiale.

Questa è l'architettura utilizzata per costruire sistemi cloud, in quanto è richiesta una grande capacità di computazione, tolleranza ai guasti e possibilità di ridimensionamento, pur minimizzando i costi di messa in essere per il provider.

2.1.4 Cluster low cost ed edge computing

Nei sistemi distribuiti, per portare la computazione più vicino alla fonte dei dati, vengono adottate soluzioni su scala ridotta situate sull'edge. Mentre all'interno dei datacenter i cluster cloud sono composti da macchine server tradizionali, quando ci si sposta al di fuori di un ambiente dedicato alla computazione il discorso cambia. È infatti preferibile utilizzare macchine dai consumi, dimensioni e costi contenuti.

Un discorso simile si può fare anche quando si costruiscono cluster casalinghi, tipicamente finalizzati all'apprendimento. In questo caso vengono tipicamente

utilizzati *Single board computer* (SBC) low cost, che permettono la realizzazione di cluster rispettando i vincoli descritti in precedenza.

2.1.5 Raspberry Pi

Raspberry Pi è uno degli SBC più popolari in ambito consumer. Per un costo e un volume molto contenuto offre la maggior parte delle feauture di un pc tradizionale. La versione più recente, il Raspberry Pi 4B, offre fino ad 8GB di RAM ed un processore quad core con architettura ARM a 64 bit.

2.1.6 ARM

ARM, acronimo di *Advanced RISC machine* è una famiglia di processori RISC, sviluppati sotto licenza di Arm ltd. È ormai diventata una delle architetture più diffuse, essendo utilizzata in tutti gli ambiti: da server a smartphone e prodotti IoT [33].

I principali vantaggi di questa architettura sono: le elevate performance a parità di consumi con x86-64, il supporto ad architetture multicore anche eterogenee, come *Big.LITTLE* e l'utilizzo del set di istruzioni ridotto permette di ridurre le dimensioni del processore e quindi anche quelle del device che lo utilizza.

2.2 Infrastruttura per il deployment di applicazioni

In questa sezione verranno descritte le opzioni disponibili per effettuare il deployment di applicazioni e le principali tecnologie a supporto.

2.2.1 Storia

Per capire le ragioni per cui si è arrivati all'infrastruttura odierna è opportuno menzionare l'evoluzione del deployment di applicazioni software nel tempo.

Deployment tradizionale

In passato le applicazioni venivano eseguite direttamente su server fisici. Questo portava a problemi di allocazione di risorse, infatti nel caso di più applicazioni in esecuzione sullo stesso server, era possibile che una singola applicazione consumasse la maggior parte delle risorse, compromettendo le performance delle altre.

Per ovviare a questo problema era necessario utilizzare una macchina separata per un ogni singola applicazione, ma questo portava al problema opposto: non

essendo sfruttate a pieno tutte le risorse disponibili risultava costoso mantenere più server.

Deployment virtualizzato

Per ovviare al problema è stata introdotta la virtualizzazione. Questa tecnica permette di eseguire su una singola macchina fisica più sistemi operativi virtualizzati, allocando ciascuna *macchina virtuale* (VM) alla singola applicazione.

Questo permette un uso più efficiente delle risorse hardware a disposizione, in cambio però di un maggiore overhead computazionale dovuto al fatto di dover gestire un sistema operativo completo per ogni macchina virtuale.

Deployment su container

Utilizzati particolarmente nelle architetture a *microservizi*, dove la classica applicazione monolitica viene suddivisa in più cocci che interagiscono fra di loro, tipicamente realizzati tramite container. I container sono simili alle macchine virtuali, ma con vincoli di isolamento più rilassati, in quanto condividono il kernel con il sistema operativo dell'host. Questo li rende particolarmente leggeri, assieme ad altri vantaggi [30]:

- La possibilità di creare immagini durante la fase di build dell'applicazione consente il disaccoppiamento tra essa e l'infrastruttura su cui viene esguita;
- Le immagini sono portabili e consentono il testing in locale, essendo possibile eseguire i container allo stesso modo di come verranno eseguiti sul cloud;
- L'immutabilità delle immagini permette il continuos delivery e il rollback delle immagini;

2.2.2 Hypervisor e piattaforme di virtualizzazione

In ambito enterprise, per la gestione di grandi numeri di macchine virtuali, vengono utilizzati software dedicati alla gestione di esse, chiamati *Hypervisor*. Esistono due tipologie di hypervisor, distinte da come interagiscono con l'hardware sottostante:

Hypervisor di tipo 1 Viene eseguito direttamente sull'hardware, chiamato anche hypervisor nativo o bare metal. Essendo eseguito in sostituzione del sistema operativo della macchina host, permette di ridurre l'overhead e garantisce performance migliori;

Hypervisor di tipo 2 Viene eseguito su un sistema operativo convenzionale come applicazione e si interfaccia con l'hardware attraverso l'emulazione. Utilizzato principalmente su PC privati da utenti singoli. Alcuni esempi sono VMware Workstation e Oracle VirtualBox [15].

Principali alternative

Avendo posto il focus sull'ambito enterprise, ci si è concentrati sugli hypervisor di tipo 1. Di seguito vengono elencate le principali alternative:

VSphere/ ESXi Sviluppato da VMware, è una delle soluzioni commerciali più diffuse. VSphere è il nome dell'ambiente completo di virtualizzazione, che unifica attraverso l'interfaccia web la gestione di cluster interi dedicati a questo scopo. ESXi è invece la denominazione dell'hypervisor che viene eseguito sulle macchine. Esso si basa su un kernel proprietario, vmkernel, che espone tre interfacce: una diretta con l'hardware, una per i sistemi guest e una per la console di gestione.

All'interno di VSphere viene inoltre utilizzato un file system appositamente sviluppato per la gestione di immagini di disco per macchine virtuali, VMFS [63];

Hyper-V Sviluppata da Microsoft, questa tecnologia permette di avere accesso ad un hypervisor di tipo 1 in ambiente Windows. Richiede quindi necessariamente un sistema operativo host Windows;

XenServer Sviluppato da Citrix, è basato sull'hypervisor Xen che, come ESXi, adotta un approccio a microkernel. Al contrario di ESXi e Hyper-V però gode del supporto della community opensource e di svariate aziende, tra cui principalmente Citrix;

Proxmox Proxmox è una piattaforma opensource, basata sulla distribuzione di Linux Debian, che permette la gestione di due tecnologie di virtualizzazione: KVM e LXC.

La prima consente la creazione di macchine virtuali su Linux, godendo dell'integrazione all'interno del kernel, agendo essenzialmente come hypervisor di tipo 1, sfruttando il supporto nativo dell'hardware alla virtualizzazione. LXC invece utilizza i container di Linux, permettendo l'esecuzione di più sistemi linux virtualizzati con un ridotto consumo di risorse.

Il sistema può essere gestito dalla GUI web o attraverso le API RESTful. Sono disponibili feature avanzate di clustering con funzionalità HA e di gestione dello storage e dei backup [55].

2.2.3 Container

Un container è un meccanismo leggero utilizzato per isolare processi, limitandone l'accesso solamente alle risorse che gli vengono allocate. Permette l'esecuzione di svariate istanze di applicazioni sullo stesso host con un consumo ridotto di risorse e con visibilità limitata solo al loro contesto di esecuzione.

Container image

Una *container image* è un file binario, scaricabile da un *registry server*, contenente il necessario per l'esecuzione di un container. Può essere composta da più layer, definiti dallo standardi Open Container Initiative (OCI), a cui aderiscono la maggior parte dei container engine.

Container engine

Il *container engine* è il componente software che interagisce con l'utente, scarica le immagini e, attraverso il *container runtime*, esegue i container. I principali container engine sono: Docker, RKT, CRI-O ed LXD.

Container runtime

Il *container runtime* si occupa dell'esecuzione dei container, interagendo con il kernel per istanziare i processi e gestisce gli aspetti di isolamento dal sistema operativo.

Container registry

Un *registry server* è un servizio di storage di file dedicato alla gestione di immagini di container. Quando un'immagine non è disponibile in locale, viene scaricata dal registry. È possibile utilizzare registry pubblici messi tipicamente a disposizione dagli sviluppatori dei vari container engine o utilizzarne di privati self-hosted [37].

Docker

Docker è uno dei container engine più diffusi. È im grado di gestire tutto il lifecycle di un container, dalla definizione, alla build e all'esecuzione, permettendo anche l'orchestrazione tramite docker-compose e docker swarm [2.2.5].

2.2.4 Container orchestration engine

Una volta sviluppata un'applicazione con una architettura basata su container è necessario uno strumento per metterla in esecuzione.

Mentre sarebbe tranquillamente possibile a livello teorico una gestione "manuale" dei container necessari attraverso script fatti ad hoc, questo non è per nulla adatto ad uno scenario di produzione. Infatti, mentre per un'architettura monolitica poteva esser considerata accettabile una gestione manuale dell'applicazione, che avrebbe risieduto su una singola VM o server fisico, gestire un elevato numero di container a mano sarebbe molto oneroso e complesso.

Per far fronte a questa necessità, occorre utilizzare un *container orchestration engine*(COE). Esso si occupa di tutti gli aspetti di gestione del lifecycle e dell'interazione dei container componenti l'applicazione, in maniera automatica.

Le feature più importanti offerte da un coe sono la gestione del provisioning e del deployment dei container, la gestione dell'allocazione di risorse ad essi, lo scaling e rimozione di container in base alle necessità e il servizio di load balancing, che permette di replicare i container.

Di seguito verranno brevemente descritte le principali alternative.

2.2.5 Alternative

Docker Swarm La modalità swarm di Docker permette di gestire un cluster di engine docker in esecuzione su macchine differenti in maniera unificata [7];

Apache Mesos Mesos nasce come strumento di gestione cluster, non limitandosi solo alla gestione di container, ma supportando vari tipi di workload, come analytics e big data. Permette di interagire con il datacenter come se fosse un pool di risorse [1].

Kubernetes Orchestratore di container open source creato da Google.

Sì è ritenuto opportuno utilizzare Kubernetes come COE, essendo la soluzione più matura e diffusa.

2.2.6 Kubernetes

Kubernetes è una piattaforma di orchestrazione di container. Rappresenta un framework per l'esecuzione di applicazioni distribuite, incorporando tutti gli aspetti di gestione di una applicazione costrutita da più container.

Fornisce servizi di *service discovery* e di *load balancing*, gestione automatizzata del rollout e rollback di un'applicazione e aggiunge funzionalità di *self-healing*, risolvendo automaticamente possibili crash dei container gestiti.

Architettura

Un cluster Kubernetes è composto da due tipologie di macchine: *master* e nodi *worker*. I nodi master eseguono il *control plane*, che gestisce i nodi worker. I nodi worker ospitano i pod al cui interno vengono eseguite le applicazioni, descritti in Sezione 2.2.6.

Componenti del control plane

Il control plane prende le decisioni sul cluster, come lo scheduling container e reagisce agli eventi.

I componenti del control plane possono essere eseguiti su qualsiasi macchina del cluster, ma tipicamente viene creata una macchina master dedicata sulla quale non verrano eseguiti container utente. Nel caso di installazione HA può essere distribuito fra più nodi master.

Di seguito vengono descritti i componenti principali del control plane:

kube-apiserver Componente del control plane che espone le API del cluster per l'interazione dall'esterno;

etcd Database distribuito consistente e HA, che mantiene tutte le informazioni relative al cluster;

kube-scheduler Assegna ai pod creati un nodo su cui essere eseguiti, in base a parametri come: requisiti di risorse, vincoli e possibili interferenze;

kube-controller-manager Esegue i processi controller. Essi si occupano del tenere d'occhio lo stato del cluster, effettuando cambiamenti al fine di raggiungere lo stato desiderato;

cloud-controller-manager Componente che implementa logica specifica relativa al cloud su cui è in esecuzione il cluster, permettendo l'interazione del cluster con esso.

Componenti nodo

I componenti nodo sono eseguiti su ogni macchina del cluster, gestendo i pod sotto il controllo del control plane, attraverso i seguenti componenti:

kubelet Componente responsabile dei pod in esecuzione sul nodo worker. Ne verifica esecuzione e salute e reagisce agli eventi di scheduling inviati dal control plane;

kube-proxy Proxy di rete che permette l'implementa il concetto di *service*, gestisce le connessioni in entrata e in uscita da un pod;

Container runtime Componente software che gestisce ed esegue effettivamente i container. Sono supportate varie tipologie, aderenti allo standard Kuberneets CRI (Container Runtime Interface) [25].

Componenti addon

I seguenti componenti non sono strettamente necessari al funzionamento del cluster, vengono eseguiti nel namespace `kube-system` ed aggiungono feature importanti al cluster:

DNS Uno degli addon più utili, fornisce il servizio di DNS all'interno del cluster, mantenendo record per i *service* di kubernetes;

Dashboard Permette di gestire tutti gli aspetti del cluster attraverso una GUI web, dalle applicazioni al cluster stesso;

Resource monitoring Tiene traccia del consumo di risorse dei container in esecuzione.

Concetti pricipali

Di seguito verranno definit i concetti principali relativo all'utilizzo di Kubernetes.

Namespace

I *namespace* forniscono un modo per poter isolare gruppi di risorse all'interno di un cluster. Questo è necessario in cluster con molti utenti per suddividere fra i vari utenti le risorse. Sono utili anche per evitare conflitti di nome tra risorse durante le fasi di test, per esempio creando un namespace per lo sviluppo ed uno per la produzione. Kubernetes fornisce quattro namespace di default:

default Collocazione di default per oggetti senza namespace;

kube-system Namespace per oggetti creati dal sistema di Kubernetes;

kube-public Accessibile da tutti gli utenti, utilizzato per risorse disponibili all'intero cluster;

kube-node-lease Contiene gli oggetti *lease* necessari all'invio di *heartbeat* per monitorare la salute dei nodi [26].

Risorse

All'interno di kubernetes lo stato del sistema è specificato attraverso *risorse*, che descrivino lo stato desiderato che il cluster poi cercherà di ottenere. Sono fornite al cluster tipicamente sotto forma di file YAML, attraverso `kubectl apply`. Di seguito sono descritte le principali risorse.

Pod Un *Pod* è l'unità più piccola deployabile all'interno di Kubernetes. Rappresenta un set di container, con storage e rete condivisi, corredati da una specifica che spiega come eseguire i container [27].

Workload Un *workload* è un'applicazione in esecuzione su kubernetes, può essere composta da uno o più pod che cooperano fra loro. Per semplificare la gestione del set pod è possibile specificare *workload resources*, che configurano i controller per la gestione dei pod. Di seguito le tipologie fornite di default:

Deployment e ReplicaSet Utili per gestire applicazioni stateless, dove ogni pod nel *deployment* è rimpiazzabile senza conseguenze;

StatefulSet Serve ad eseguire pod che possiedono uno stato;

DaemonSet Definisce pod utili al funzionamento del cluster, garantisce l'esecuzione dei pod su ogni nodo;

Job e CronJob Rappresentano task che eseguono fino al completamento per poi interrompersi. *CronJob* definisce un task ricorrente [31].

Service Un *service* è un modo astratto per esporre un'applicazione in esecuzione su un gruppo di pod alla rete. I pod sono risorse non permanenti, dato che un deployment li potrebbe distruggere e creare dinamicamente, quindi nonostante ogni pod riceva un indirizzo IP, è necessario un modo per tenere traccia degli indirizzi dei pod in esecuzione. Il service effettua il load balancing tra i pod associati ad esso, rendendo trasparente l'accesso alle repliche [28].

Configurazione Nel caso sia necessario fornire dati di configurazione ai pod, è possibile utilizzare un oggetto *ConfigMap*, permettendo il disaccoppiamento di configurazioni relative all'ambiente dall'immagine del container. Nel caso si tratti di dati sensibili è opportuno invece definire una risorsa di tipo *secret*.

Role-based access contol (RBAC) *RBAC* è un modo per regolamentare l'accesso a risorse in base ai ruoli degli utenti nel sistema. Attraverso *Role* e *ClusterRole* è possibile definire i permessi associati ad un ruolo, con il primo in base al namespace, con il secondo per tutto il cluster. Attraverso un *RoleBinding* o un *ClusterRoleBinding* i permessi vengono associati ad un utente o un set di utenti [29]. Un *ServiceAccount* permette di associare un'identità ad un pod, in modo da permettergli di interagire con l'API con i permessi giusti.

Scalabitità in Kubernetes

Uno dei principali vantaggi offerti da Kubernetes è la possibilità di reagire ai cambiamenti, adattando le risorse di conseguenza. Sono stati definiti tre principali modalità di scaling all'interno di Kubernetes, riassunte di seguito [3].

Horizontal scaling All'interno di un'applicazione (in genere stateless), potrebbe essere necessario incrementare il numero di pod, per far fronte ad incrementi di utilizzo. Questo è possibile attraverso l'horizontal pod autoscaler, che controllando le metriche di utilizzo delle risorse dai pod, determina se ridurre o aumentare il numero di essi.

Vertical scaling Di base Kubernetes effettua un overcommitting sulle risorse disponibili su un nodo, assumendo che l'utilizzo tipico di un pod sia inferiore dal limite richiesto. Il vertical autoscaler permette di modificare dinamicamente, in base all'effettivo utilizzo, questi limiti, permettendo un utilizzo ottimale delle risorse disponibili su un nodo.

Cluster scaling Un cluster autoscaler ha un compito simile a quello dell'horizontal pod autoscaler, ma riferito ai nodi che compongono il cluster. Esso infatti monitora gli eventi di pod non schedulabili, espandendo in questo caso il cluster istanziando un nuovo nodo, oppure rileva nodi non utilizzati e li dealloca. Richiede però il supporto dalla parte piattaforma sottostante per l'allocazione e la distruzioned di risorse.

2.2.7 K3s

K3s è una distribuzione fully compliant di Kubernets sviluppata da Rancher, concepita per l'utilizzo su hardware con risorse limitate, ottimizzata per architettura ARM. Le principali differenze con una installazione tradizionale sono:

- Sostituzione del backend di storage etcd con sqlite3, più leggero;
- Sicurezza di default;

- Aggiunta di default di funzionalità utili: storage contoller locale, load balancer di servizi, controller Helm e l'ingress controller Traefik;
- Incapsulamento dell'operazione dei componenti del control plane in un singolo binario;
- Containerd come container runtime;
- Minimizzazione delle dipendenze esterne, tramite inclusione delle dipendenze necessarie nel binario [56].

Viene distribuito sotto forma di un singolo binario, permettendo la scelta del ruolo del nodo fornendo l'argomento server o agent all'esecuzione.

Funzionalità integrate

Di seguito verranno elencate le funzionalità aggiuntive offerte di default dalla distribuzione k3s. Sono inclusi vari servizi di rete [57]:

CoreDNS Server DNS per il service discovery;

Traefik ingress controller Reverse proxy HTTP e load balancer;

Klipper load balancer Service load balancer.

Per quanto riguarda la gestione di volumi e storage persistenti ne è possibile la creazione attraverso il Local Path Provisioner di Rancher.

2.3 OpenStack

OpenStack può essere considerato come un Sistema Operativo per il cloud, utilizzato per amministrare grandi pool di risorse di computazione, archiviazione e networking all'interno di un cluster di computer. Viene utilizzato per la creazione di private e public cloud. L'architettura di OpenStack consiste in una suddivisione in componenti, dove ognuno fornisce uno specifico servizio, accessibile tramite una API con un metodo di autenticazione comune.

2.3.1 Componenti principali

Di seguito vengono descritti i servizi di OpenStack rilevanti al lavoro svolto in questa tesi, partendo dai servizi core. Le relazioni principali fra di essi sono riassunte in Figura 2.1.

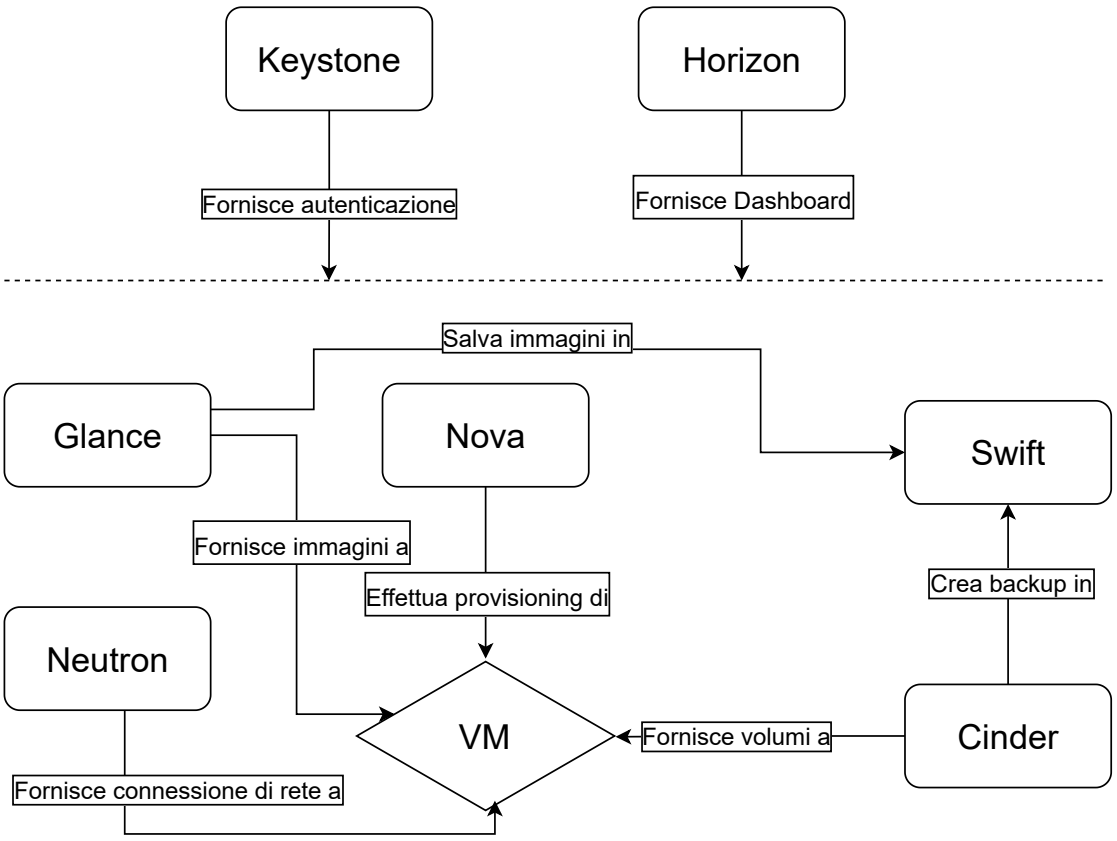


Figura 2.1: Relazioni tra i componenti principali di OpenStack

Compute: Nova

Nova è il servizio che fornisce la possibilità di effettuare provisioning di *server virtuali* (chiamate istanze compute), utilizzando l'*hypervisor* messo a disposizione dall'host.

È a sua volta composto da più servizi, che possono essere dislocati su server differenti, coordinandosi fra loro con messaggi RPC:

DB Database SQL per storage di dati: tiene traccia delle istanze disponibili,di quelle in uso e delle reti disponibili;

API Componente che riceve le richieste HTTP dall'esterno e comunica con gli altri componenti;

Scheduler Determina su che host eseguire ogni specifica istanza;

Compute Gestisce la creazione o distruzione delle macchine virtuali attraverso le API degli hypervisor;

Conductor Gestisce richieste che richiedono coordinazione, come build e resize, fa da proxy per l'accesso al DB;

Placement Tiene traccia dell'uso dei provider di risorse [51].

Image Service: Glance

Il servizio di immagini Glance permette agli utenti di caricare e usufruire *data asset* da usare con altri servizi, in particolare *immagini* di macchine virtuali e *definizioni di metadati*. Le immagini gestite da Glance possono essere salvate semplicemente sul file system o su servizi di object storage [44].

Dashboard: Horizon

Horizon è l'implementazione di riferimento per la *dashboard* di OpenStack. Fornisce un'interfaccia web estensibile con plugin, utilizzata per poter interagire graficamente con i servizi offerti dal cluster.

Identity Service: Keystone

Keystone si occupa dell'*autenticazione* all'interno di OpenStack, rilasciando i token per l'accesso alle API. All'interno di OpenStack è possibile definire utenti(tenant), a cui è possibile dare l'accesso ai vari servizi forniti dal private cloud, impostando quote di utilizzo.

È internamente suddiviso in più servizi, generalmente utilizzati in maniera combinata dal frontend:

Identity Si occupa fornire la validazione delle credenziali di autenticazione e di gestire i dati di *utenti e gruppi*. Un utente è inteso come il consumatore individuale dell'API, mentre un gruppo è una collezione di utenti;

Resource Fornisce dati su *progetti e domini*. Ogni risorsa in OpenStack deve essere di proprietà di un *progetto* specifico e i *domini* sono un contenitore di progetti, utenti e gruppi;

Assigment Permette di ottenere informazioni su *role* e *role assignment*. Un *role* determina le autorizzazioni che l'utente possiede, mentre il *role assignment* è una tripla di *role*, *risorsa* e *identità*;

Token Valida e gestisce i *token* necessari ad autenticare le richieste dopo che l'utente si è autenticato con successo;

Catalog Utilizzato per l'*endpoint discovery* [47].

Networking: Neutron

Neutron è il componente che permette la creazione di infrastruttura di rete virtuale come reti, sottoreti e router secondo un approccio *NaaS* (Networking as a Service).

Gestisce inoltre anche l'accesso alle reti esterne (fisiche) permettendo l'allocazione di indirizzi IP speciali, denominati *floating IP*, disponibili sulla rete esterna, a *porte* sulla rete interna. Permette così a una macchina virtuale collocata su quest'ultima rete e associata alla specifica porta di comunicare con l'esterno. Le reti interne sono reti virtuali connesse direttamente alle VM e sono connesse fra loro e alle reti esterne attraverso router virtuali.

Include anche un servizio di *firewall*, attraverso la definizione di *security groups* a cui le VM appartengono, che regolamentano porte e tipo di traffico possibile per esse [50].

Storage Prima di descrivere i servizi relativi allo *storage* disponibili all'interno di OpenStack è opportuno distinguere gli approcci possibili.

La metodologia di archiviazione più semplice è quella basata su *file*, in cui per accedere ad un dato è necessario conoscere il percorso.

Un altro approccio è il *block storage*, dove i dati vengono invece suddivisi in blocchi e collocati dove più opportuno, e nel momento in cui è necessario accedere al file, i blocchi vengono recuperati e riassemblati. Questo approccio è più performante dello storage come file, non dovendosi affidare ad un unico percorso, ed è utilizzato per file di grandi dimensioni.

L'*object storage* invece suddivide i file in unità discrete, poi conservati in un unico repository. Rispetto allo storage a blocchi, dove ogni blocco è identificato

solo dal suo indirizzo, il file è mantenuto come un unico oggetto, permettendo un vasto uso di *metadati* associabili ad esso, utilizzati assieme all'identificativo per recuperare il file. È però meno vantaggioso quando vengono effettuate delle modifiche ai file, in quanto l'intero oggetto dovrà essere riscritto sul disco [59].

Block Storage: Cinder

Cinder è il servizio di *block storage* utilizzato per fornire *volumi* persistenti alle VM all'interno di OpenStack. Oltre ai volumi permette la creazione di *snapshot*, copie di sola lettura di un volume in uno specifico istante temporale e *backup*, copie di archivio di un volume.

I moduli che compongono il sevizio sono:

cinder-api Autentica e indirizza le richieste al servizio block storage;

cinder-scheduler Determina il volume service a cui affidare ogni singola richiesta, utilizzando uno scheduler round-robin o il più sofisticato *Filter scheduler*, che permette di filtrare i servizi disponibili;

cinder-volume Interagisce con i device di backend del servizio block storage. Supporta vari driver di back-end, il driver offerto di default è basato su *LVM*;

cinder-backup Permette il backup di un volume di block storage ad un oggetto *object storage* [46].

Object Storage: Swift

Swift è un object store *highly availabile*, *distribuito* ed *eventually consistent*. La sua architettura distribuita senza un punto di controllo centrale permette una maggiore scalabilità e ridondanza [53].

Provisioning bare metal: Ironic

Ironic è il componente di OpenStack che permette il provisioning di macchine *bare metal*, descritto in maggiore dettaglio in Sezione 2.4.3. Espone un'interfaccia al servizio Compute che permette di gestire macchine fisiche come macchine virtuali.

2.3.2 Servizi secondari

Di seguito sono descritti i progetti non facenti parte del core di OpenStack che però sono rilevanti al contesto di questa tesi.

Orchestration: Heat

Consente di orchestrare applicazioni cloud attraverso un formato di template dichiarativi.

Container Orchestration Engine Provisioning: Magnum

Permette la creazione di *container orchestration engine*, in particolare di cluster *Kubernetes*, *Docker Swarm* e *Apache Mesos*, descritti in dettaglio in Sezione 2.2.4. Si appoggia su *Heat* per orchestrare VM basate su un'immagine apposita al fine di creare il cluster. I cluster vengono definiti attraverso *cluster template* che ne definiscono la struttura e consentono ripetibilità nella creazione di essi [49]. Magnum offre quindi all'utente finale un cluster managed, occupandosi degli aspetti di gestione del cluster, al pari dei servizi EKS, GKE, AKS.

2.3.3 Servizi di supporto ad OpenStack

Pur non facendo parte del progetto, questi servizi sono parte integrante e necessari al corretto funzionamento di un cluster OpenStack.

Database SQL

Buona parte dei servizi OpenStack utilizzano un database relazionale per immagazzinare informazioni, tipicamente la scelta ricade su un cluster MariaDB/Galera o MySQL.

Message queue

Per coordinare le operazioni fra servizi e condividere informazioni di stato viene utilizzata una coda di messaggi, la scelta della maggior parte delle distribuzioni è RabbitMQ.

High availability services

In un deployment *Highly Available* (HA) di OpenStack vengono utilizzati i servizi *Keepalived* e *HAProxy*. Il primo si occupa di fornire indirizzi IP virtuali, mentre il secondo viene utilizzato come *load balancer* software per i servizi di backend [45].

2.3.4 Metodi di deployment

Negli anni sono stati sviluppati vari tool per facilitare il deploy di OpenStack e per scegliere quello più adatto vanno considerate varie precondizioni:

Use case In base allo scopo per cui viene costruito il cluster variano i requisiti del tool di deployment. Un proof of concept ha requisiti meno stringenti di un setup da utilizzare in produzione;

Lifecycle management I tool utilizzati per il deployment del cluster verranno in seguito sfruttati per la gestione e l'aggiornamento di esso;

Progetti OpenStack Non tutti i progetti di deployment supportano tutti i componenti di OpenStack;

Provisioning bare metal La gestione dell'installazione del sistema operativo su host bare metal è parte del processo di deployment del cloud, alcuni tool integrano la gestione di questo aspetto;

Vendor lock-in Esistono progetti di deployment sviluppati da aziende esterne come Canonical (2.3.4, 2.3.4) ma si corre il rischio di dipendere dalle scelte architetturali dell'azienda e di essere impossibilitati ad uscire dal loro ecosistema [36].

Di seguito verranno brevemente descritte le principali opzioni disponibili per il deployment di OpenStack

Installazione manuale

L'installazione manuale, basandosi sulla ricca documentazione disponibile è sicuramente l'opzione migliore per imparare a fondo come funziona OpenStack, come si configura e come le parti interagiscono fra di loro.

Lo svantaggio è che non essendo per niente automatizzata, è necessario configurare e installare manualmente ogni singolo servizio che compone il cluster. Inoltre, durante l'operazione del cluster, è più complessa la gestione e manutenzione di esso e la correzione di eventuali errori di configurazione.

Devstack

Devstack è un insieme di script volti a creare rapidamente un ambiente OpenStack funzionante e completo, tipicamente composto da un solo nodo, su una singola macchina (virtuale).

Permette di avere accesso a tutte le feature più recenti, dato che i vari progetti vengono installati direttamente dai repository contenti il codice sorgente. Questo porta anche instabilità, essendo il codice meno testato delle release ufficiali.

È principalmente utilizzato come ambiente di testing funzionale di OpenStack e pertanto non è concepito come installazione persistente, ma per un uso effimero, rendendolo inadatto per un ambiente di produzione.

Microstack

Microstack è una distribuzione di OpenStack sviluppata da Canonical, volta a una rapida installazione su una singola macchina, racchiudendo i componenti e le dipendenze necessarie in un unico pacchetto *Snap*. Al momento comprende Glance, Horizon, Keystone, Neutron e Nova e supporta il clustering in modalità beta [5].

Nonostante sia pubblicizzato per le applicazioni *edge* e di *micro-cloud*, la distribuzione per architettura ARM è al momento meno sviluppata rispetto a quella per x86 e non supporta ancora il clustering.

OpenStack Charms

Questo progetto, sviluppato sempre da Canonical permette il deploy del cloud utilizzando *JUJU*. I servizi sono suddivisi in *Charm*, messi in relazione fra loro attraverso *Bundle*, descritti in un *Modello*, gestiti e deployati da un *Controller*, interagendo con un *Cloud*.

Il cloud sottostante mette a disposizione risorse compute e storage, consumate dal controller per deployare i Charm. Nonostante JUJU supporti i cloud provider più diffusi, nel caso si voglia effettuare il deploy su bare metal l'unica opzione supportata è MAAS [21]. In questo caso il fattore vendor lock-in è molto presente, oltre alle questioni descritte in precedenza, si è costretti a rimanere nell'ecosistema dell'azienda madre per tutti gli aspetti del cluster, come il sistema operativo base (Ubuntu) e le scelte architetturali, che vengono dettate dal *Modello*. Mentre tecnicamente è possibile riconfigurare il modello di un deployment le relazioni possibili fra i componenti sono implicitamente dettate da quello che i *Charm* utilizzati espongono. Inoltre non sono presenti *Charm* per tutti i componenti OpenStack, per esempio *Magnum* non è ancora supportato in maniera stabile [52].

OpenStack Ansible

Utilizza Ansible (descritto in **??**) per effettuare il deploy di un cluster OpenStack basato su container linux (LXC).

Bifrost

Permette l'utilizzo di Ironic in maniera standalone, sfruttando Ansible per eseguire l'installazione di una immagine base su hardware già conosciuto, in base ad un file di inventory. È pensato per facilitare la costruzione da zero dell'infrastruttura, imponendo meno requisiti operazionali possibile [43].

Kolla-Ansible

Kolla-Ansible è un progetto di deployment sviluppato dalla community OpenStack che utilizza Ansible (descritto in Sezione 2.5.3) per effettuare il deployment di *Kolla*. Kolla è un progetto che ha lo scopo di costruire immagini docker che racchiudono i progetti di OpenStack.

Il deployment viene eseguito a partire da un host con ha il ruolo di controller, tipicamente esterno al cluster, che attraverso Ansible e i file di configurazione necessari effettua l'installazione dei container di Kolla.

Il file di inventory specifica quali host fanno parte del cluster e gli assegna un ruolo, in base ad esso verranno installati i servizi corrispondenti. Attraverso il file `globals.yml` è possibile specificare la configurazione del cluster: quali servizi includere ed alcuni parametri di configurazione relativi. Eventuali configurazioni più avanzate non coperte dal file globale possono essere specificate secondo lo standard di configurazione di OpenStack, posizionati nella directory `/etc/kolla/config` sul host controller [48]. Le password sono gestite attraverso il file `passwords.yml`, generabile con il comando `kolla-genpwd`.

I principali vantaggi di *Kolla-Ansible* sono la sua flessibilità e le opzioni di configurazione, mantenendo comunque una certa semplicità utile all'apprendimento, fornendo una configurazione di default funzionante. È possibile gestire anche il lifecycle del cluster, infatti aggiornando i file di configurazione e di inventory il cluster viene riconfigurato sfruttando le potenzialità di Ansible e non è necessario rieseguire il deploy da capo. Non supporta però il deployment di host bare-metal.

Kayobe

Acronimo di *Kolla on Bifrost*, questo progetto completa Kolla e Kolla-Ansible con Bifrost, aggiungendo così la possibilità di effettuare il deployment di host bare-metal senza l'utilizzo di altri tool esterni, integrando anche la gestione di device fisici di rete ed aggiungendo un'interfaccia a linea di comando.

TripleO

Acronimo di *OpenStack On OpenStack*, questo progetto sfrutta un'istanza di OpenStack, definita *undercloud*, per deployare una seconda istanza completa, direttamente su bare metal, denominata *overcloud*. Questo approccio, anche se molto complesso, permette di avere una completa gestione del lifecycle del cluster, adatto a deployment su larga scala all'interno di datacenter. Inoltre sono sfruttate esclusivamente le API di OpenStack, senza introdurre tool esterni [54].

2.4 Provisioning di host bare metal

Anche la gestione dell'hardware è un aspetto su cui sono stati fatti progressi a livelli di automazione. Mentre in passato era necessario andare di macchina in macchina con un supporto di installazione ad effettuare l'installazione del sistema operativo, ad oggi esistono tecnologie e tool basati su di queste, che permettono di effettuare queste operazioni completamente da remoto.

Inoltre, in ambito cloud, la possibilità di avere a disposizione l'intera macchina offre alcuni vantaggi all'utente finale:

- L'assenza di un Hypervisor riduce l'overhead di computazione, permettendo di sfruttare a pieno le risorse a disposizione;
- In alcune applicazioni avere accesso all'hardware direttamente è necessario, particolarmente per hardware particolare;
- Viene evitato l'effetto *noisy neighbor*, ossia una singola VM che consumando troppe risorse del server limita le performance delle altre VM in esecuzione.

2.4.1 Tecnologie chiave

Il provisioning automatizzato di host bare metal attraverso la rete è reso possibile da un insieme di tecnologie, descritte di seguito.

Preboot Execution Environment (PXE)

Lo standard PXE permette al BIOS e alla scheda di rete (NIC) di un PC di effettuare il *bootstrapping* del sistema operativo dell'host attraverso la rete. Il bootstrapping è definito come il processo di caricare il sistema operativo in memoria in modo da farlo eseguire al processore [42];

Dynamic Host Configuration Protocol (DHCP)

Il protocollo DHCP viene utilizzato per la configurazione automatica dei parametri di rete degli host, come indirizzi IP delle interfacce. In questo caso l'host che effettua il boot attraverso PXE ottiene dal server DHCP, opportunamente configurato, l'indirizzo del PXE boot server e il percorso del file di bootstrap da scaricare.

Network Bootstrap Program (NBP)

Il file NBP è l'equivalente dei bootloader come *GRUB* utilizzati per il boot da disco locale. La sua responsabilità principale è recuperare e caricare il kernel del sistema operativo in memoria (via rete).

Trivial File Transfer Protocol (TFTP)

Il protocollo TFTP è un protocollo di trasferimento file che implementa le funzionalità di base di FTP. Viene utilizzato dal client PXE per scaricare il file NBP in base alle informazioni fornite dal server DHCP. La sua semplicità di implementazione lo rende adatto allo scopo in quanto deve essere contenuto all'interno del firmware presente sulla scheda di rete.

Negli anni sono stati introdotti strumenti per il boot da rete più avanzati, come per esempio iPXE [20], generalmente eseguiti con una tecnica di *chainloading*, dove l'implementazione PXE della scheda di rete scarica iPXE tramite TFTP e lo esegue. Questo approccio permette di supportare protocolli più avanzati, veloci e sicuri, come per esempio HTTP.

Baseboard Management Controller (BMC)

In ambito server è sempre presente un controller per la gestione *out-of-band*. Questo controller possiede una sua interfaccia di rete riservata, attraverso il quale è possibile configurare i parametri del BIOS di un host e controllarne accensione e spegnimento da remoto, in maniera indipendete dal sistema operativo. Uno degli standard più diffusi è IPMI.

Wake-on-LAN, uno standard che permette di accendere un PC attraverso l'invio di un pacchetto speciale, seppur supportato dalla maggior parte delle schede di rete non è considerabile come BMC. Questo perchè non garantisce l'accensione del PC e non è possibile gestire lo spegnimento al di fuori del SO, non potendo quindi agire in caso di crash.

2.4.2 MAAS

MAAS (Metal as a Service) è un tool di provisioning di server bare-metal, sviluppato da Canonical. Sfrutta lo stack PXE ed immagini effimere per permettere il provisioning automatico di host bare metal.

La funzionalità di MAAS è suddivisa in due tipologie di controller: un *region controller* e uno o più *rack controller*, descritti di seguito.

Region controller

Il *region controller* si occupa sia di comunicare con l'utente, attraverso l'interfaccia web e l'API REST, sia di gestire i *rack controller* presenti nel sistema. Mantiene lo stato delle macchine registrate sistema nel database PostgreSQL e fornisce ai rack controller le immagini da utilizzare per il provisioning. Inoltre si occupa degli aspetti più autoritativi della gestione di rete, come fornire il servizio di DNS e di proxy HTTP con funzionalità di caching.

Rack controller

Il rack controller gestisce le macchine contenute nel *server rack* in cui è posizionato, occupandosi di fornire servizi che richiedono una grande larghezza di banda. Comunica con esse attraverso il *fabric* di rete, a cui sono connessi gli elementi del rack e che esso gestisce direttamente. In particolare fornisce i servizi necessari al provisioning dei server: DHCP per assegnare un indirizzo IP e rendere possibile il boot da rete, server TFTP e HTTP per fornire file NBP e immagini da avviare agli host. Interfacciandosi con il BMC degli host ne gestisce accensione e spegnimento. Mantiene una cache locale dei file più voluminosi per migliorare le performance, ma non mantiene uno stato indipendente al di fuori di ciò che è necessario per la comunicazione con il region controller [34].

Il region controller si può quindi considerare come l'entità responsabile per un singolo datacenter, mentre il rack controller gestisce un singolo armadio di server. Per deployment di piccole dimensioni è possibile collocare entrambe le tipologie di controller sulla stessa macchina.

Rete

La gestione della rete di provisioning è un aspetto fondamentale per il funzionamento di MAAS, di seguito verranno illustati i concetti principali.

Il concetto di livello più alto è quello di *fabric*, che grossolanamente corrisponde ad una connessione fisica di rete e può contenere più sottoreti. All'interno di un fabric è possibile definire *VLAN*, a cui appartengono le sottoreti, che possono essere gestite o meno da MAAS. Una VLAN (Virtual LAN) è un modo per creare reti logicamente separate che si appoggiano sulla stessa infrastruttura fisica.

Per poter operare a pieno MAAS ha bisogno di avere completo controllo su una sottorete, agendo da gateway per essa e da server DHCP. In una sottorete gestita da MAAS è possibile riservare alcuni indirizzi IP in modo da allocarli ad usi non strettamente collegati al provisioning degli host. Attraverso il riservamento dinamico vengono allocati gli indirizzi IP che possono essere utilizzati per le fasi di enlisting, commissioning e deploy dei nodi. In una sottorete non gestita è solo possibile riservare indirizzi IP utilizzabili da MAAS, che però devono essere resi disponibili dal servizio gestore, in quanto in questo caso MAAS non fornisce i servizi di DHCP.

DHCP Snippet Per personalizzare il comportamento del server DHCP integrato, è possibile specificare opzioni aggiuntive, che possono essere applicate sia a livello globale, che a livello di subnet o di singolo host. Essendo il servizio di DHCP fornito da dhcpd, le eventuali modifiche dovranno aderire al suo standard

di configurazione, in quanto verranno inserite direttamente nel file `dhcpd.conf`, che determina il comportamento del DHCP.

Lifecycle di un nodo

All'interno di MAAS ad ogni nodo è sempre associato uno stato, attraverso gli stati principali si può descrivere il lifecycle dei nodi all'interno del sistema [35]:

New Una macchina che effettua il boot via PXE su una rete gestita da MAAS viene automaticamente registrata;

Commissioning In questa fase viene fatto l'inventario di RAM, CPU, dischi, schede di rete e altri device messi a disposizione dall'host in esame, registrando tutto nel sistema;

Ready Quando il commissioning ha esito positivo una macchina viene definita *Ready*, questo include una corretta configurazione delle credenziali del BMC per poter permettere a MAAS di controllarne accensione e spegnimento;

Allocated Una macchina *Ready* può essere allocata ad un utente, che potrà configurarne le schede di rete ed i dischi;

Deploying In seguito è possibile installare in maniera completamente automatica un sistema operativo sulla macchina, applicando le configurazioni scelte in precedenza;

Releasing Quando la macchina non è più necessaria, può essere di nuovo rilasciata nel pool di macchine disponibili.

2.4.3 OpenStack Ironic

Ironic fa parte del progetto di private cloud open source OpenStack, descritto in dettaglio in 2.3, nascendo come driver bare metal per il componente compute di OpenStack. Ad oggi è possibile utilizzarlo in maniera indipendente per la gestione di di macchine bare metal.

I componenti principali di Ironic sono i seguenti:

API RESTful Riceve e processa le richieste dai client, inviandole al conductor;

Ironic Conductor Gestisce lo stato nel sistema dei nodi, ricevendo istruzioni dall'API via RPC. Effettua le operazioni di provisioning, deploy e decommissioning. Gestisce l'accensione e spegnimento dei nodi attraverso il BMC, interfacciandosi con il *driver* corrispondente;

Driver hardware Si occupa della gestione dell'hardware in maniera specifica in base al vendor del BMC;

Ironic Agent Viene eseguito temporaneamente sul nodo target durante la fase di deploy, effettua le operazioni preliminari e scarica l'immagine da installare;

Ironic Client Interfaccia a linea di comando per l'API di Ironic;

Ironic Web Client Plugin per la dashboard di Openstack.

Inoltre si appoggia a risorse esterne come gli altri servizi di OpenStack: un database SQL per mantenere lo stato dei nodi e una coda di messaggi per la comunicazione RPC fra i componenti [41].

Confronto con MAAS

Rispetto a MAAS permette una gestione più libera dell'effettivo sistema operativo che viene installato sulle macchine, ma così facendo richiede la creazione manuale delle immagini da installare, senza offrire una gestione automatica come MAAS.

MAAS inoltre permette una gestione del sistema attraverso un'interfaccia web, mentre Ironic per poter utilizzare la sua interfaccia web richiede la presenza della dashboard di OpenStack, che a sua volta richiede la presenza di altri servizi, complicando l'installazione. Quindi in un'installazione stand-alone di Ironic si è limitati alla sola interfaccia a linea di comando.

Questa funzionalità non è fondamentale, però risulta utile la possibilità di monitorare le fasi del processo di deployment degli host in maniera grafica. Inoltre, essendo concepito per la gestione di server all'interno di un datacenter, pone come requisito alle macchine gestite, oltre al supporto per PXE, la presenza di un BMC tra quelli supportati. MAAS offre una gestione molto più automatica dell'ambiente in cui viene effettuato il deploy, gestendo anche alcuni aspetti di rete, ed effettuando automaticamente l'enlisting nel sistema delle macchine.

Considerati questi fattori si è scelto di utilizzare MAAS in quanto offre una gestione più completa di tutta l'infrastruttura, unita ad una maggiore semplicità d'uso.

2.5 Gestione automatizzata dell'infrastruttura

La gestione automatizzata dell'infrastruttura consiste nell'utilizzare tecnologie che effettuano tutti i task di gestione necessari al controllo di hardware, software e dell'infrastruttura di rete con un ridotto intervento umano.

Questo è dettato da una continua crescita della complessità e della dimensioni dell'infrastruttura necessaria a supportare le applicazioni moderne. Sarebbe molto

difficile continuare gestire manualmente tutte le operazioni tipiche di messa in essere, configurazione e manutenzione dei sistemi.

L'introduzione di automazione alle suddette operazioni consente di riguadagnare controllo e visibilità d'insieme sull'infrastruttura [17].

2.5.1 Pet vs Cattle

La possibilità di gestire in maniera automatica l'intero lifecycle ha portato ad un cambiamento alla visione che l'amministratore ha dei server, virtuali o fisici, che gestisce.

In passato una macchina era trattata come un animale domestico, prestandogli particolari cure ed attenzioni. Questo perchè dovendo effettuare gran parte delle operazioni di installazione, configurazione e manutenzione manualmente era richiesto un grande sforzo di tempo, ed era più conveniente "salvare" un'installazione che dava problemi più che reinstallare il sistema operativo e configurare la macchina da capo.

Oggi invece, grazie alla diffusione dell'automazione in questo ambito, è possibile trattare i server come una mandria, senza preoccuparsi troppo della salute del singolo, in quanto facilmente rimpiazzabile da una nuova macchina in maniera automatica.

2.5.2 Terraform

Terraform è un tool di *Infrastructure as code* open source creato da HashiCorp.

Permette di gestire l'infrastruttura IT descrivendola utilizzando un linguaggio di configurazione di alto livello (HCL) che descrivono la topologia desiderata. È in grado di gestire sia componenti di livello più basso come istanze compute che elementi di alto livello come record DNS.

Infrastruttura Immutabile

Uno dei vantaggi principali di Terraform è la creazione di infrastruttura immutabile. Secondo questo approccio, ad ogni cambiamento dell'ambiente la vecchia configurazione è rimpiazzata con una nuova configurazione e viene effettuato il reprovisioning dell'infrastruttura.

Questo aiuta a prevenire il fenomeno del *configuration drift*, che si può verificare con una gestione mutabile dell'infrastruttura, dove lo stato effettivo non corrisponde più a ciò che è stato specficato nella configurazione [19].

Listato 2.1: Elementi base della sintassi di HCL

```
resource "openstack_networking_floatingip_v2" "fip_agent
   " {
  pool  = var.floating_ip_pool
}

<TIPOLOGIA BLOCCO> "<ETICHETTA BLOCCO>" "<ETICHETTA
   BLOCCO>" {
  # Corpo del blocco
  <IDENTIFICATORE> = <ESPRESSIONE> # Parametri
}
```

HashiCorp Configuration Language (HCL)

Una configurazione Terraform è un documento scritto nel linguaggio di configurazione di Terraform (HCL) che specifica come gestire una collezione di infrastrutture.

Nel listato 2.1 è fornito un esempio di risorsa e sono descritti gli elementi base del linguaggio di configurazione. Il contenitore principale è il *blocco*, a cui è sempre associato una *tipologia* e zero o più *etichette*. Il blocco rappresenta un oggetto, tipicamente una risorsa. All'interno del corpo del blocco possono essere presenti dei *parametri*, che associano ad un *identificatore* dei valori. Le *espressioni* sono dei valori che possono essere anche riferimenti a variabile o combinazione di altri valori.

L'ordine dei blocchi non determina l'ordine di applicazione, viene determinato in base alle relazioni implicite ed esplicite fra di essi [13].

Provider

Un *provider* è un plugin che permette l'interazione con servizi, piattaforme cloud o API, mettendo a disposizione risorse e/o fonti di dati. Permette a Terraform di interagire con il servizio al posto dell'utente e di di effettuare il provisioning delle risorse specificate nel file di configurazione.

Risorsa La *risorsa* è l'elemento più importante all'interno di Terraform. Permette di descrivere gli elementi dell'infrastruttura e la loro configurazione.

Workflow di provisioning

La gestione dell'infrastruttura attraverso Terraform si può descrivere in tre step principali, descritti di seguito. Questi step vengono eseguiti al termine della progettazione e all'inzializzazione dell'ambiente Terraform, attraverso il comando `terraform init`.

Pianificazione Prima di effettuare il deploy dell'infrastruttura occorre decidere un piano di azione. Eseguendo il comando `terraform plan` viene valutata una configurazione di Terraform, in modo da determinare lo stato desiderato delle risorse specificate.

Questo stato viene poi comparato allo stato effettivo delle risorse, determinando le modifiche necessarie. L'output finale del comando consiste nelle modifiche necessarie per raggiungere lo stato desiderato.

È possibile salvare l'output della fase di comunicazione in un file, che poi potrà essere dato in input al comando `terraform apply`, descritto di seguito.

Applicazione Il comando `terraform apply` esegue la pianficazione allo stesso modo di `terraform plan`, ma una volta determinato il piano procede add applicanrlo, istanziando le risorse attraverso le API del provider. È possibile fornire in input in file di pianificazione creato nella fase precedente.

Distruzione Con il comando `terraform destroy` è possibile effettuare il deprovisioning dell'infrastruttura creata attraverso Terraform. Attraverso i dati dello stato sono determinati quali oggetti reali corrispondono alle risorse descritte nella configurazione [12].

2.5.3 Ansible

Ansible è un tool di automazione di infrastruttura informatica, che permette la gestione configurazioni, deployment di applicazioni e orchestrazione. È un progetto open source sponsorizzato da Red Hat. La principale differenza con Terraform, che utilizza un approccio puramente dichiarativo, è la possibilità di specificare la configurazione in maniera procedurale.

Di seguito ne verranno descritte le principali caratteristiche e funzionalità.

Architettura

Una delle caratteristiche che distingue Ansible dagli altri tool con funzionalità e scopi è la sua architettura. Opera infatti in maniera *agentless*, secondo un modello *push*, non richiedendo quindi l'installazione di software addizionale sulle macchine per renderle gestibili.

L'unico requisto per poter gestire una macchina attraverso Ansible è l'accesso attraverso SSH (per Linux e UNIX) o WinRM (per Windows). Inoltre non richiede accesso da amministratore, appoggiandosi a metodi di escalazione di privilegi come sudo o su quando necessario.

Questo aspetto rende il tool intrinsecamente più sicuro, infatti così solo chi possiede le credenziali di accesso ad una macchina è in grado di gestirla da remoto. Inoltre, il fatto di operare secondo un modello push fa sì che le macchine gestite vedano esclusivamente il codice necessario alla configurazione, chiamato modulo, senza essere a conoscenza di come vengono configurate altre macchine.

Essendo *agentless* Ansible non utilizza risorse di computazione delle macchine gestite nei momenti in cui le sta attivamente gestendo [58].

Playbook

Un *playbook* è la definizione, in formato YAML, dello stato desiderato del sistema. È composto da una o più *play*, composta da *task* da eseguire sugli host bersaglio, definiti nell'*inventory*. Durante l'esecuzione Ansible controlla lo stato del sistema e se lo stato non corrisponde a ciò che è descritto dal playbook, effettua le opportune modifiche [14]. Un playbook viene eseguito fornendolo in input al comando `ansible-playbook`.

Questo approccio idempotente permette l'applicazione di configurazioni in maniera ripetuta senza side effect e con velocità, evitando di eseguire azioni non necessarie.

I task possono essere incapsulati in unità riusabili chiamati *role*, minimizzando la ripetizione delle configurazioni.

Capitolo 3

Obiettivi

In questo capitolo saranno descritti i principali obiettivi di questo progetto di tesi, considerando gli strumenti introdotti precedentemente nel Capitolo 2.

L'obiettivo principale della tesi è di valutare la possibilità di creare un private cloud utilizzando hardware a basso costo con architettura ARM, e su di esso costruire un cloud privato, su cui mettere in esecuzione applicazioni che sfruttino le feature offerte dal cloud sottostante.

Lo scopo non è di creare infrastruttura pronta ad essere usata in produzione, ma di ottenere una piattaforma su cui è possibile sperimentare. Infatti, mentre è possibile provare gratuitamente i principali cloud pubblici, c'è sempre il rischio di eccedere nell'utilizzo e di incorrere in spese inaspettate o di non poter accedere a particolari feature solo a pagamento. L'avere un cloud a propria disposizione, senza vincoli di utilizzo, consente di studiare le possibilità offerte da esso e di capirne a fondo l'architettura.

Un altro obiettivo è di automatizzare il più possibile tutti gli aspetti di deployment dell'infrastruttura, attraverso strumenti di IaC, con lo scopo di rendere versionabile e replicabile il lavoro prodotto.

3.1 Identificazione dell'hardware

Come prima cosa sarà necessario identificare l'hardware da utilizzare per la realizzazione fisica del cluster. Dovranno essere selezionati dispositivi di computazione, di rete e di archiviazione. Nonostante l'obiettivo sia di utilizzare dispositivi low cost, l'hardware selezionato dovrà essere in grado di eseguire i compiti desiderati in maniera soddisfacente, garantendo una buona compatibilità col software da eseguire. Un ottimo candidato per questo scopo è il Raspberry Pi uno dei più popolari SBC disponibili sul mercato consumer. Basandosi esso su architet-

tura ARM, quindi dovrà essere attentamente valutato ogni aspetto di possibile incompatibilità, adattando eventualmente il software pensato per x86.

3.2 Gestione di un bare metal cloud

Per semplificare la messa in essere del cluster ci si è posto l'obiettivo di gestire automaticamente la fase di installazione del sistema operativo e di configurazione delle macchine bare metal.

Il tool più adatto allo scopo è MAAS, che dovrà essere adattato alla gestione delle macchine Raspberry Pi, attraverso anche l'introduzione di un BMC realizzato ad hoc.

3.2.1 Installazione automatizzata di MAAS

Per garantire la maggior replicabilità possibile del setup, si è deciso di automatizzare l'installazione dell'host che eseguirà MAAS. Per raggiungere questo scopo sarà necessario utilizzare strumenti di Infrastructure as code, in particolare Terraform per la parte di creazione della macchina virtuale sull'hypervisor e Ansible per l'installazione e la configurazione delle opzioni di MAAS.

3.3 Deployment del cluster Openstack

Usando come base le macchine messe a disposizione da MAAS, verrà costruito un cluster OpenStack. Dovrà essere identificata l'architettura più consona all'hardware a disposizione, scegliendo una distribuzione ottimale dei servizi. Il cluster realizzato dovrà permette l'aggiunta di nuove macchine in maniera semplice. Per effettuare l'operazione di deployment dovrà essere identificato il tool più adatto.

3.4 Deployment automatico del cluster Kubernetes

Sfruttando la piattaforma IaaS messa a disposizione da OpenStack verrà costruito in maniera automatica un cluster Kubernetes. Questo sarà possibile attraverso servizi offerti direttamente da OpenStack o in alternativa costruendo il cluster attraverso tool di IaC che si interfacciano sul cloud.

3.5 Scaling automatico del cluster kubernetes

Al fine di sfruttare al meglio l'elasticità messa a disposizione dal private cloud, ci si pone l'obiettivo di implementare all'interno del cluster un sistema di scaling automatico dei nodi, istanziando nuove macchine virtuali quando il cluster kubernetes richiede nuovi nodi.

Capitolo 4

Testing preliminare

Vista la complessità e stratificazione del sistema che si ha intenzione di costruire, descritta in dettaglio nel Capitolo 5, si è ritenuto opportuno effettuare alcune prove preliminare su alcuni componenti critici. Questo ha l'obiettivo di identificare l'alternativa giusta da adottare, visto che è possibile scegliere più strade per il raggiungimento dello stesso scopo.

4.1 Testing funzionale OpenStack

Una volta costruita l'infrastruttura si è ritenuto opportuno effettuare dei test funzionali preliminari al fine di valutare la fattibilità del deploy di OpenStack sull'hardware a disposizione. Le prove sono state effettuate sia utilizzando Devstack che utilizzando Kolla-ansible come metodi di deploy. Questi test hanno evidenziato che il modello da 4GB di RAM di Raspberry Pi non è adatto al ruolo di controller all'interno del cluster.

Infatti il carico sul nodo controller è molto più oneroso, utilizzando la distribuzione Kolla-ansible, su di esso sono in esecuzioni 31 container relativi ad OpenStack, con un consumo di RAM medio di 5.8GB, rispetto al carico su un nodo compute, dove 18 container consumano 1.58GB di RAM. È ovvia la conclusione che il modello da 4GB non è adatto alla funzione di nodo controller.

Un altro parametro osservato è il carico di sistema, verificabile con il comando `uptime`, esso rappresenta il livello di utilizzo del processore, dove un carico di 1 corrisponde al 100% di utilizzo, superando questo valore le performance del sistema iniziano a degradare a causa dell'aumento di processi in attesa di tempo CPU. Nel caso di sistemi multicore l'utilizzo massimo corrisponde al numero di core (Verificabili con il comando `grep processor /proc/cpuinfo | wc -l`).

Durante i test si sono registrati carichi di sistema di oltre 40 dovuti alle operazioni di swap necessarie a sopperire alla mancanza di memoria RAM. Di fatto

il sistema era inutilizzabile, in quanto quasi nella totalità delle volte i comandi eseguiti con la CLI di OpenStack andavano in timeout e la dashboard web non riusciva a superare la fase di login.

Con la sostituzione del nodo controller da 4GB con un modello da 8GB si è immediatamente notato un miglioramento delle perfomance e non sono stati riscontrati particolari problemi di performance e usabilità.

4.2 Verifica funzionalità Magnum

Una volta confermata la possibilità di ottenere un cluster OpenStack funzionante sull'hardware scelto, si è deciso di effettuare una seconda verifica preliminare, sul progetto Magnum. Esso infatti permetterebbe di risolvere molto semplicemente e in maniera nativa l'obiettivo di creare un cluster Kubernetes in maniera automatica.

Per creare un cluster con Magnum è necessario prima bisogna definire un template di cluster, dove vengono specificati i parametri di base del cluster, come l'immagine da usare per i nodi, il flavor dei nodi e il driver di rete. In seguito è possibile istanziare il cluster, specificando il numero di nodi desiderato.

Per una prima prova quindi ci si è affidati alla documentazione di Magnum. Dopo aver installato la CLI specifica per Magnum si è creato un template utilizzando come base l'immagine di fedora CoreOS 34 per ARM, utilizzando il comando seguente:

```
openstack coe cluster template create k8s-template \
--image "coreos34" \
--external-network public \
--dns-nameserver 8.8.8.8 \
--master-flavor m1.small \
--flavor m1.small \
--coe kubernetes \
--volume-driver cinder \
--network-driver flannel \
--docker-volume-size 40
```

In seguito lo si è istanziato con il seguente comando.

```
openstack coe cluster create test \
--cluster-template k8s-template \
--master-count 1 \
--node-count 2 \
--keypair mykey
```

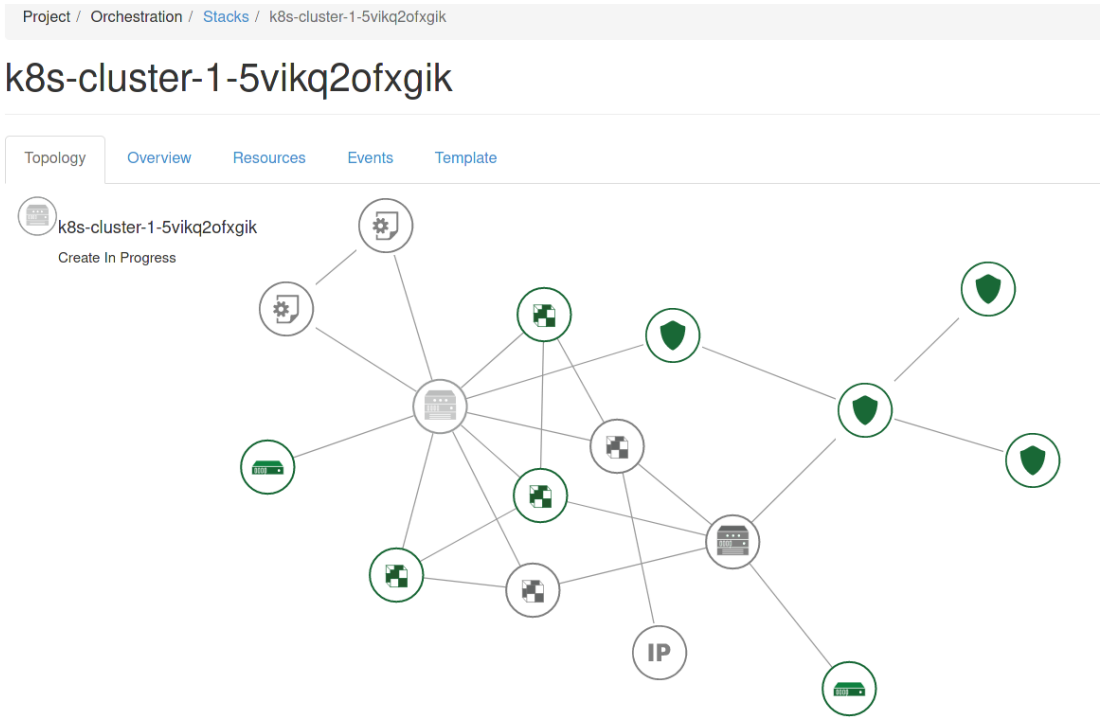


Figura 4.1: Rappresentazione grafica del cluster in costruzione offerta da Heat.

Monitorando il progresso attraverso l'interfaccia grafica di Heat, a cui Magnum si appoggia per istanziare le risorse, si è prima verificato la corretta creazione di rete, gruppi di sicurezza e macchine virtuali, come mostrato in Figura 4.1. In seguito si è notato che il processo si bloccava sulla creazione del nodo master, e facendo login su questo nodo si è potuto constatare che questo era dovuto all'utilizzo dei container errati.

Magnum infatti usa per il deploy di Kubernetes dei container Docker, che vengono scaricati da un registry gestito dal team di OpenStack. In questo registry sono presenti esclusivamente immagini buildate per architettura amd64, quindi incompatibili con arm.

Per tentare di risolvere questo problema si è provato a reperire ed hostare le immagini necessarie, buildate per l'architettura giusta, su DockerHub. Si è poi creato un nuovo template ed istanziato un altro cluster, specificando tra le opzioni il nuovo registry.

Con questi accorgimenti si è riusciti a portare a termine il deploy del cluster, almeno secondo Magnum. Infatti nonostante il cluster Kubernetes risulti creato con successo, non è effettivamente utilizzabile.

Verificando con `kubectl -n kube-system get pods` lo stato del sistema, mostrato in Listing 4.1, si può notare che tre pod sono in stato di errore, mentre

Listato 4.1: Stato del cluster alla fine dell'esecuzione di Magnum.

```
NAME                                          READY   STATUS              RESTARTS   AGE
calico-kube-controllers-59f6dc8ddb-wbdc6      0/1     Pending             0          21h
calico-node-5l2qb                             1/1     Running             0          20h
calico-node-lqrtm                             0/1     Running             0          21h
coredns-85594b6b4f-5g76t                      1/1     Running             0          21h
coredns-85594b6b4f-mtwfc                      1/1     Running             0          21h
csi-cinder-controllerplugin-0                 0/5     ContainerCreating   0          21h
dashboard-metrics-scraper-55c699dc9d-rlp8t    0/1     Pending             0          21h
k8s-keystone-auth-bkhrj                       0/1     CrashLoopBackOff    249        21h
kube-dns-autoscaler-5f67697df8-62cd7          0/1     Pending             0          21h
kubernetes-dashboard-5c88f49f9d-fpbnv         0/1     Pending             0          21h
openstack-cloud-controller-manager-j58lq      0/1     CrashLoopBackOff    252        21h
```

altri quattro in attesa. In dettaglio, il pod `csi-cinder-plugin` non è in grado di montare il volume relativo ai certificati di sicurezza, essendo configurato un path errato, avendo dovuto usare una versione più vecchia dei container. I pod `k8s-keystone-auth` e `openstack-cloud-controller-manager`, relativi all'integrazione con OpenStack, sono bloccati in un loop di crash, di cui però non è stata trovata la causa, non riuscendo ad accedere ai log. Questi pod bloccati bloccano a loro volta la creazione del resto dei pod di sistema, ponendo il cluster in uno stato inutilizzabile.

Non riuscendo a trovare una soluzione a questa problematica, è stato necessario non utilizzare Magnum per la gestione di Kubernetes in questo progetto e di affidarsi ad un altro metodo di deployment.

Capitolo 5

Progettazione del cluster

In questo capitolo verranno descritti gli aspetti relativi alla progettazione, tra cui l'architettura del cluster ed i metodi utilizzati per costruirlo.

5.1 Hardware

Visto l'obiettivo di realizzare un cluster a basso costo la scelta delle macchine è ricaduta su Raspberry Pi 4. Essi sono uno degli SBC più diffusi sul mercato e meno costosi, offrendo comunque un feature set completo tra cui: CPU ARM quad core, 2/4/8GB di RAM, connettività USB2/USB3/USBC, WiFi 802.11ac e gigabit ethernet, Bluetooth 5.0 [9].

Vista l'impossibilità, determinata dalla modalità scelta per il deploy (descritta in seguito), di utilizzare come supporto di archiviazione per il sistema operativo la scheda microsd fornita assieme ai Raspberry si è dovuto valutare altre opzioni di archiviazione. Come supporto di archiviazione per i Raspberry si è quindi scelto di utilizzare Hard disk meccanici da 2.5", in quanto facilmente reperibili usati e dalle permance sufficienti. Questa soluzione ha consentito così di mantenere ancora più basso il costo di costruzione del cluster.

Per il collegamento dei dischi sono stati utilizzati adattori da SATA ad USB3, che non necessitano di alimentazione esterna, garantendo così performance ottimali per i dischi e un ridotto ingombero. Per alimentare i Raspberry si è scelto di utilizzare l'alimentatore da 5V a 3A incluso nel kit a disposizone, in quanto sufficientemente potente.

È stato inoltre necessario reperire uno switch e cavetteria varia,come i cavi ethernet per la connessione LAN dei Raspberry, un cavo micro HDMI-HDMI per il debugging degli host e cavi dupont per la connessione del BMC ai pin GPIO dei Raspberry. Per realizzare il BMC è stato scelto un microcontrollore ESP32, in

Tabella 5.1: Tabella riassuntiva dei componenti fisici del sistema

Oggetto	*Quantità*
Raspberri Pi 4B 8GB	1
Raspberri Pi 4B 4GB	3
Hard disk 500GB	4
Cavo USB3 - SATA	4
Cavo RJ45 Cat 5e	4
Switch Gigabit 8 porte Cisco SG-200	1
Alimentatore USB-C 15W Raspberry Pi	4
Microcontrollore ESP32	1
Connettori dupont	12
Modulo base stampato in 3d	4
Modulo copertura stampato in 3d	1
Tastiera USB, cavo micro HDMI-HDMI, scheda microsd	1

quanto soddisfa il requisito di possedere una interfaccia di rete ed abbastanza pin GPIO.

Per mantere ordinato e ridurre l'ingombero del cluster si è scelto di utilizzare un case per i raspberry. Per ridurre i costi è stato scelto di realizzarlo con una stampante 3d reperendo online il modello da stampare [4].

I componenti utilizzati sono riassunti in Tabella 5.1

5.2 Provisioning Raspberry Pi

Una volta identificato l'hardware si è dovuto decidere quale modalità adottare per effettuare il provisioning del sistema operativo sui Raspberry Pi. Sono state identificate due strade percorribili:

Installazione manuale Viene utilizzato il metodo "tradizionale" di installazione del sistema operativo su Raspberry Pi: si copia manualmente i file relativi al sistema operativo su ogni singola scheda, per poi inserirla all'interno delle macchine. Utilizzando questa metodologia la costruzione del cluster risulta molto più scomoda e inefficiente.

Installazione automatizzata via rete Questo approccio permette di ridurre notevolmente il tempo impiegato per la costruzione del cluster. Inoltre, offrendo la possibilità di reinstallare da remoto il sistema operativo, dopo la costruzione del cluster non è più necessario intervento fisico per riconfigura-

re le macchine. Richiede però una macchina separata che faccia da server e gestisca i Raspberry.

Visto che l'automazione è uno degli obiettivi principali di questa tesi, la scelta è ovviamente ricaduta sull'installazione automatizzata, utilizzando MAAS, descritto in dettaglio in Sezione 2.4.2.

In quanto MAAS è un applicativo orientato principalmente all'ambito server saranno necessari alcuni accorgimenti per poter gestire a pieno il provisioning dei Raspberry.

5.3 MAAS

Si è scelto di collocare entrambi i componenti di MAAS, region e rack controller, sulla stessa macchina, in quanto il setup ridotto non dovrebbe presentare criticità riguardo alla gestione di molti host e quindi controller ridondanti o su macchine separate. Per consentire il boot via rete con MAAS è stato necessario trovare una alternativa all'utilizzo del firmware PXE presente di default nella EEPROM dei Raspberry, in quanto è una implementazione non fully-compliant e pertanto non funziona correttamente con MAAS. Per poter funzionare in maniera automatica è necessario poter controllare automaticamente accensione e spegnimento, attraverso un BMC. I Raspberry, essendo questo un componente tipico dell'ambito server e che avrebbe alzato il costo del prodotto, non ne sono dotati. Per ovviare a questa mancanza sarà necessario utilizzare un BMC costruito ad hoc ed interfacciarlo con MAAS.

La soluzione ai problemi di interazione tra Raspberry Pi e MAAS proviene dal lavoro di Lorenzo Mondani [39], di cui verranno descritti i dettagli di seguito.

5.3.1 BMC

Per realizzare il BMC si è stato scelto un microcontrollore ESP32. Esso dovrà essere in grado di spegnere ed accendere il Raspberry e verificarne lo stato di alimentazione. Per una questione di semplicità si è deciso di far gestire al singolo BMC tutti i Raspberry presenti all'interno del cluster. Dovrà essere in grado di interagire via rete con il componente dedicato all'interno di MAAS, esponendo le seguenti API:

GET /rpis Ritorna un oggetto JSON contenente lo stato di alimentazione di ogni Raspberry connesso al BMC, secondo la forma {`<id>:{"power":<on,off>}`};

GET /rpis/⟨id⟩ Ritorna sotto forma di JSON lo stato di alimentazione del singolo Raspberry, analogamente al caso precedente;

PUT /rpis/⟨id⟩ Permette di cambiare lo stato di un Raspberry, specificando lo stato desiderato nel body della richiesta. Il BMC procederà poi ad effettuare l'operazione di spegnimento.

L'interazione con i Raspberry avverrà attraverso i pin GPIO, in particolare il pin GPIO3 verrà utilizzato come pulsante di alimentazione, tramite ACPI. La lettura dello stato di accensione è possibile verificando la tensione del pin GPIO14.

Power Driver MAAS

Per permettere l'utilizzo del BMC all'interno di MAAS è necessario implementare un Power Driver da utilizzare all'interno di MAAS. Esso deve implementare l'interfaccia `PowerDriver` ed interagire con il BMC attraverso gli endpoint esposti.

5.3.2 Chainloading UEFI

Per poter utilizzare PXE a pieno con MAAS sarà necessario effettuare il chainloading del firmware UEFI preparato in precedenza. Questo sarà effettuato inserendo uno snippet DHCP contenente la configurazione necessaria a scaricare dal server TFTP built-in di MAAS i file relativi a UEFI.

5.3.3 Progettazione di rete

Per permettere a MAAS di funzionare al meglio, si è deciso di collocare i Raspberry su una rete isolata. MAAS infatti opera fornendo il servizio di DHCP sulla rete da lui controllata e, vista l'impossibilità di modificare la rete principale connessa ad Internet ed il server DHCP operante all'interno di essa, è stato necessario adottare questa soluzione.

Al fine di raggiungere questo scopo è stato utilizzato uno switch Cisco SG-200, che permette la creazione di VLAN, dividendo così il traffico trasmesso sul dominio broadcast principale da quello sul secondario. Sulla VLAN 1 è stato allocato quindi il traffico della subnet 192.168.1.0/24, mentre sulla VLAN 2 è stata allocata la subnet 10.0.0.0/24, dedicata al cluster.

Per permettere l'accesso dall'esterno si è deciso di fornire alla macchina virtuale su cui viene eseguito MAAS due interfacce di rete. La prima con indirizzo IP 192.168.1.81 è connessa alla rete principale e può quindi comunicare con l'esterno ed Internet, mentre la seconda, con IP 10.0.0.2 è dedicata alla gestione del cluster, dove agisce anche da gateway. Questo è reso possibile dalla disponibilità di due interfacce di rete sulla macchina fisica in cui è in esecuzione Proxmox, l'hypervisor scelto per virtualizzare MAAS. Queste interfacce vengono condivise con le VM in esecuzione attraverso il bridging.

Nella subnet gestita da MAAS sono stati riservati due blocchi di indirizzi:

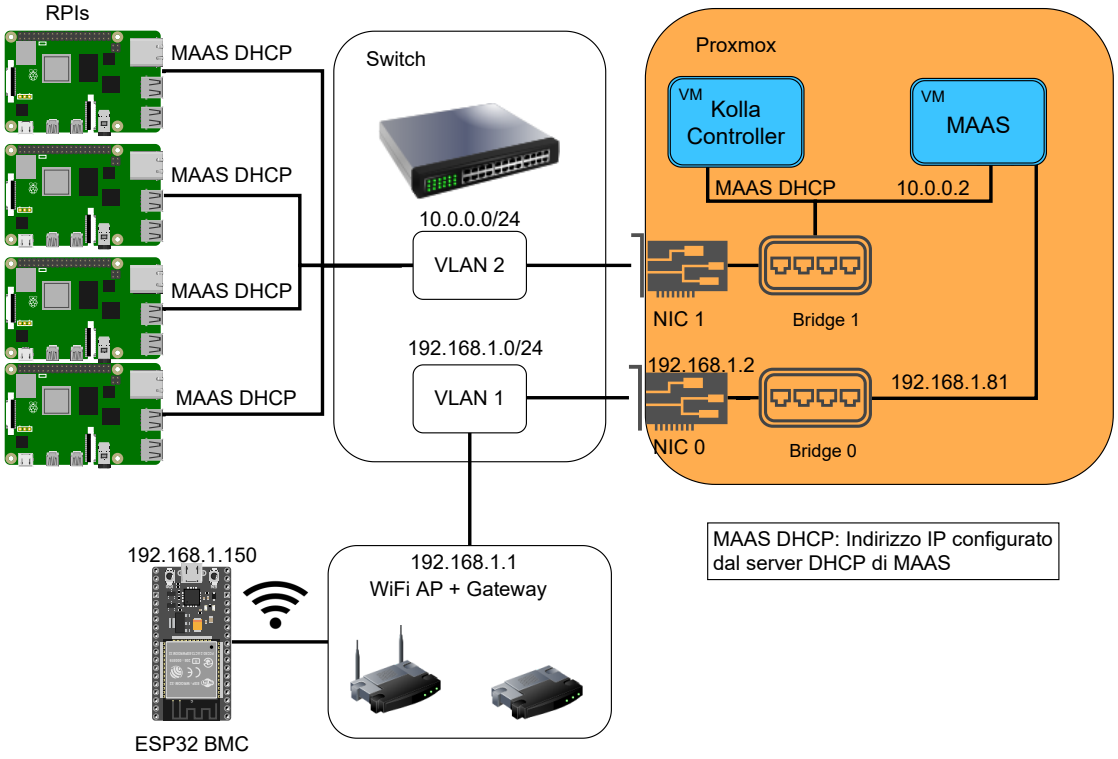


Figura 5.1: Architettura di rete del sistema

- **10.0.0.10-10.0.0.49:** Range riservato ad OpenStack, su cui verranno allocati IP virtuali e floating;

- **10.0.0.50-10.0.0.100:** Range dinamico utilizzato per le fasi di provisioning assegnabile a nodi gestiti da MAAS.

Il BMC, basato su ESP32, è connesso via Wifi alla rete wireless messa a disposizione sulla subnet 192.168.1.0/24, potendo così comunicare con il server MAAS. La macchina dedicata alla gestione di OpenStack attraverso Kolla-Ansible è allocata come VM su Proxmox e gestita da MAAS ed ha accesso unicamente alla sottorete 10.0.0.0/24.

L'architettura di rete è riassunta in Figura 5.4.

Sshuttle Per consentire l'accesso alle macchine all'interno del cluster isolato è stato utilizzato sshuttle [61], tool che permette di accedere a reti remote semplicemente avendo accesso via SSH ad un host sulla rete a cui si vuole accedere.

5.4 Progettazione cluster Openstack

In questa sezione vengono descritti i punti principali relativi alla realizzazione del cluster OpenStack. Esso segue un ciclo di release semestrale, ed al momento della progettazione la release più recente e supportata era Wallaby, quindi la scelta è ricaduta su di essa.

5.4.1 Architettura cluster

Aspetto importante della progettazione di un cluster OpenStack è la disposizione dei servizi sui vari nodi. Si è scelto di dedicare un host interamente ai servizi del control plane: autenticazione, scheduling, gestione immagini, database, coda di messaggi e dashboard. Gli altri tre nodi, denominati compute, ospiteranno i servizi compute e storage. Vi è quindi una suddivisione su più macchine di servizi che logicamente sono considerati una unica unità, come Cinder e Nova, dove la parte che si occupa dello scheduling risiede sul nodo di controllo e la parte che interagisce con il rispettivo backend è situata sul nodo compute. Alcuni servizi sono presenti su tutti i nodi, come gli agenti di rete, i servizi relativi alla High availability e il loggin centralizzato, fornito da fluentd.

La suddivisione dei servizi tra le tipologie di host è riassunta in Figura 5.2.

5.4.2 Scelta tool di deployment

Tra tutte le possibili opzioni presentate in Sezione 2.3.4 è stato scelto Kolla-Ansible. Questo tool infatti permette un grande grado di flessibilità, fornendo però dei valori di default sensati permettendo anche la gestione dell'aggiunta e rimozione di nodi da un cluster già istanziato.

Per una questione di semplicità si è scelto di non implementare un mirror su un registry in locale delle immagini Docker necessarie.

5.4.3 Configurazione nodi

Per poter installare con successo OpenStack i nodi dovranno soddisfare alcuni requisiti. Per il corretto funzionamento dei servizi di SDN è necessario avere a disposizione almeno due interfacce di rete, di cui un verrà utilizzata per la connettività esterna, mentre l'altra sarà interamente gestita dal servizio di rete. Visto che i Raspberry possiedono una sola interfaccia LAN e che quella wireless non è adatta ad un utilizzo senza configurazione, si è quindi optato per l'utilizzo di due interfacce virtuali [32]. La prima è stata inserita in bridge con l'interfaccia fisica e la seconda è stata lasciata completamente in gestione a Neutron.

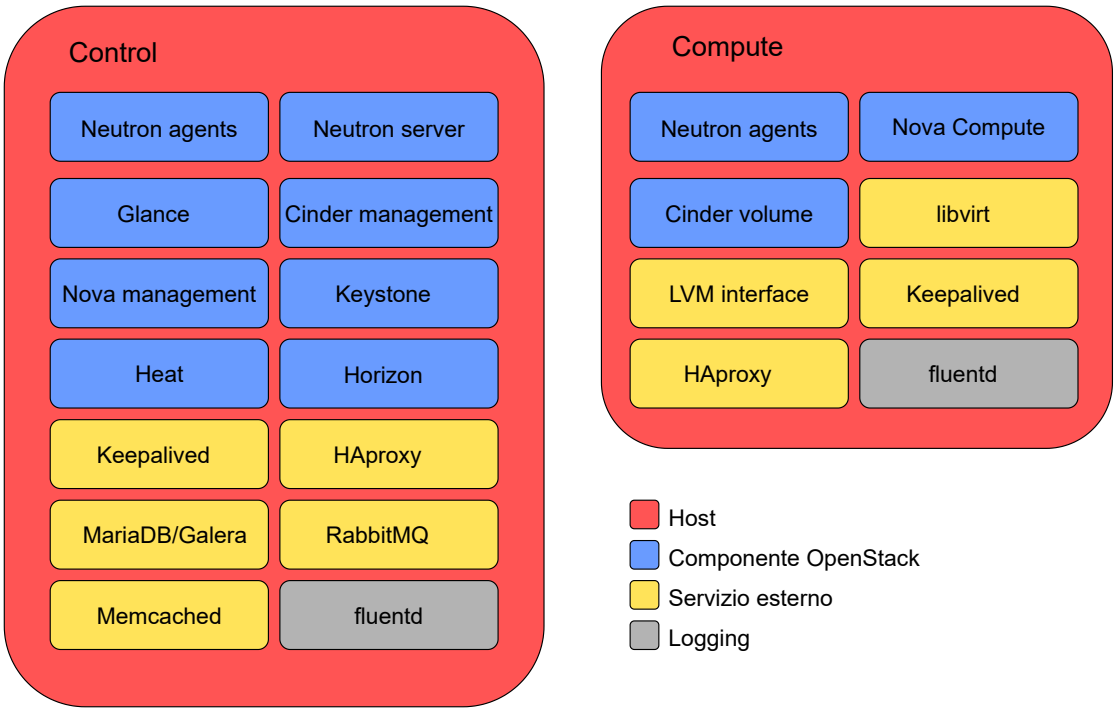


Figura 5.2: Collocazione dei servizi sui nodi.

Per applicare in maniera automatica la configurazione preliminare si è scelto di utilizzare Ansible.

5.5 Progettazione cluster Kubernetes

Essendo stata verificata l'impossibilità di utilizzare Magnum per la costruzione di Kubernetes in questo specifico ambiente, come descritto in Sezione 4.2, si è dovuta identificare una modalità alternativa per l'installazione del cluster Kubernetes.

La scelta è ricaduta sul progetto k3s, in quanto pone tra gli obiettivi principali del progetto la leggerezza e il supporto nativo ad ARM.

Sì è scelto di organizzare il deployment iniziale del cluster in due nodi, collocati su due macchine virtuali connesse ad una rete virtuale interna, collegata con un router virtuale alla rete esterna, come riassunto in Figura 5.3.

Per effettuare il deploy delle macchine virtuali componenti il cluster e la successiva installazione su di esse di k3s si è scelto di utilizzare Terraform.

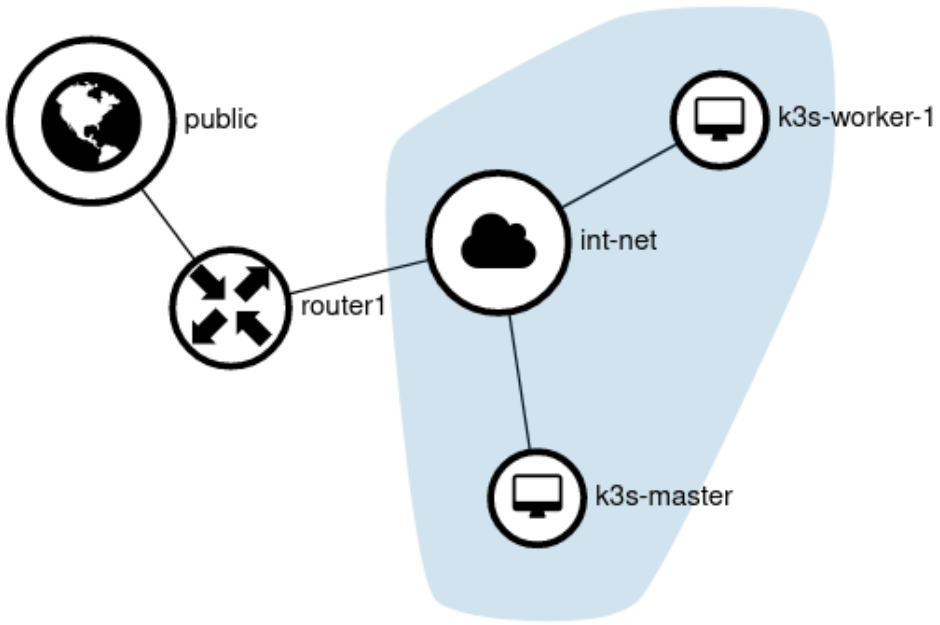


Figura 5.3: Architettura del cluster Kubernetes all'interno di OpenStack.

5.5.1 Kubernetes cluster autoscaler

Visti i problemi riscontrati con il cloud provider di OpenStack su architattura ARM, è stato necessario identificare una strada alternativa per l'autoscaler. Questo perchè non potendo utilizzare Magnum è preclusa anche la possibilità di utilizzare l'autoscaler nativo di Kubernetes, che supporta solamente Magnum all'interno di OpenStack [23].

Per realizzare lo scaling automatico è necessario poter effettuare due interazioni con il sistema:

Interazione con Kubernetes È necessario poter interrogare le API di Kubernetes, in modo da capire quando il cluster non riesce a istanziare pod per mancanza di risorse o quando ci sono nodi senza carico di lavoro, ed effettuare le operazioni di scaling;

Interazione con OpenStack L'autoscaler deve essere in grado di contattare l'API di OpenStack con il fine di istanziare una nuova macchina virtuale, su cui installare k3s e farla unire al cluster che ne necessita. Deve essere anche in grado di rimuovere le macchine virtuali superflue, quando il cluster è meno carico.

La soluzione è stata trovata nel progetto kubernetes-cluster-autoscaler [60], che rispetta entrambi i requisiti richiesti. Sarà quindi necessario estendere il progetto allo specifico caso d'uso di questo lavoro, come verrà dettagliato in Sezione 6.5.

5.6 Architettura generale del sistema

Il risultato finale della progettazione è un'architettura di sistema riassumibile in tre livelli principali:

Bare metal Questo livello viene gestito da MAAS attraverso PXE e il BMC, offrendo un servizio di Bare metal as a Service;

OpenStack Sul layer inferiore verrà costruita l'infrastruttura cloud di OpenStack, che verrà utilizzata secondo un approccio Infrastructure as a Service, mettendo a disposizione risorse compute, storage e di rete;

Kubernetes Sfruttando le risorse messe a disposizione dal layer inferiore verrà costruito un cluster Kubernetes con la possibilità di scalare automaticamente sfruttando il livello inferiore. Esso offrirà all'utente un servizio di Platform as a a Service.

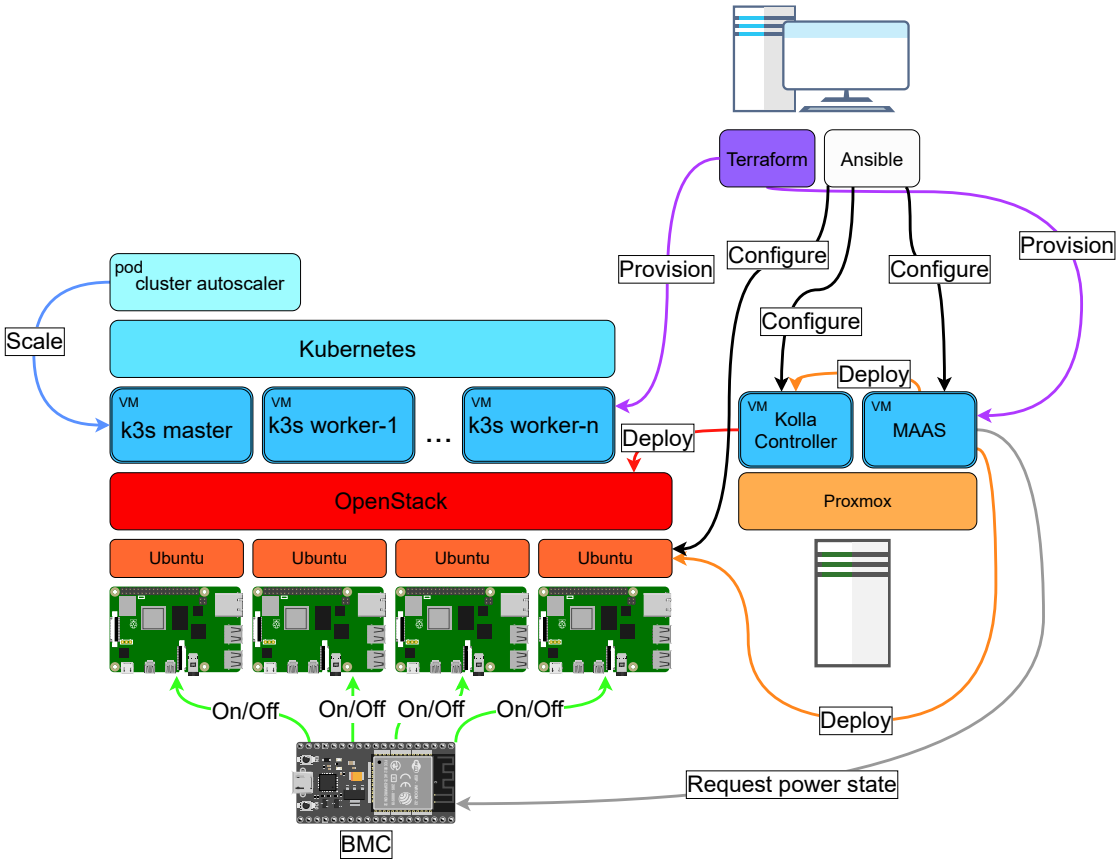


Figura 5.4: Architettura generale del sistema. Fonti modelli: [2], [62].

Scendendo più in dettaglio, per prima cosa verrà istanziata su un host Proxmox una macchina virtuale attraverso Terraform e configurata con MAAS attraverso Ansible. Il server MAAS fornirà ai Raspberry il firmware UEFI e su di essi effettuerà l'installazione e la configurazione di base del sistema operativo, sfruttando il BMC basato su ESP32. In seguito attraverso Ansible verranno configurati in maniera specifica i Raspberry e la macchina virtuale, sempre istanziata su Proxmox e gestita da MAAS, che avrà il ruolo di host da cui effettuare il deploy con Kolla. Utilizzando Kolla-Ansible verrà effettuato il deploy del cluster OpenStack sui Raspberry Pi. Attraverso Terraform verrà instanziata la necessaria infrastruttura di rete e delle macchine virtuali su OpenStack, su cui verrà installato Kubernetes. Su questo cluster Kubernetes verrà installato un cluster autoscaler, che attraverso le API di OpenStack potrà effettuare lo scale up o scale down del cluster, in base alle risorse disponibili.

in Figura 5.4 è mostrata l'architettura definita e le interazioni fra i componenti.

Capitolo 6

Costruzione del cluster

In questo capitolo verranno descritti tutti gli aspetti relativi alla messa in essere del cluster, implementando le scelte prese in fase di progettazione.

6.1 Costruzione fisica del cluster

Una volta reperito l'hardware necessario è stato possibile procedere con la costruzione fisica del cluster. Per prima cosa sono stati installati sui Raspberry i dissipatori forniti nel kit e sono stati effettuati i collegamenti tramite cavi dupont fra i pin GPIO dell'ESP32 e quelli dei vari Raspberry .

Si è inoltre proceduto con la stampa in PLA del case, utilizzando una stampante Sharebot NG, come mostrato in Figura 6.2. Una volta terminata la stampa del case, esso è stato assemblato, inserendovi i Raspberry. In seguito sono stati collegati i dischi, gli alimentatori ed è stato installato lo switch, collegando poi i cavi di rete necessari.

Per impostare le VLAN si è effettuato l'accesso alla GUI web dello switch e lì sono state configurate due VLAN separate, la prima comprendente le porte 1,2,3 e la seconda sulle restanti 5.

Il cluster completato è mostrato in Figura 6.1. Rispetto alla progettazione si è optato di aggiungere delle ventole per migliorare il raffreddamento ed è stato necessario sostituire un disco da 2.5" con uno da 3.5", nonostante le stime di facile reperibilità, che presenta funzionalità identiche ma un ingombro maggiore.

6.2 Installazione MAAS

In questa sezione vengono descritti i passaggi necessari al setup dell'ambiente e all'installazione di MAAS.

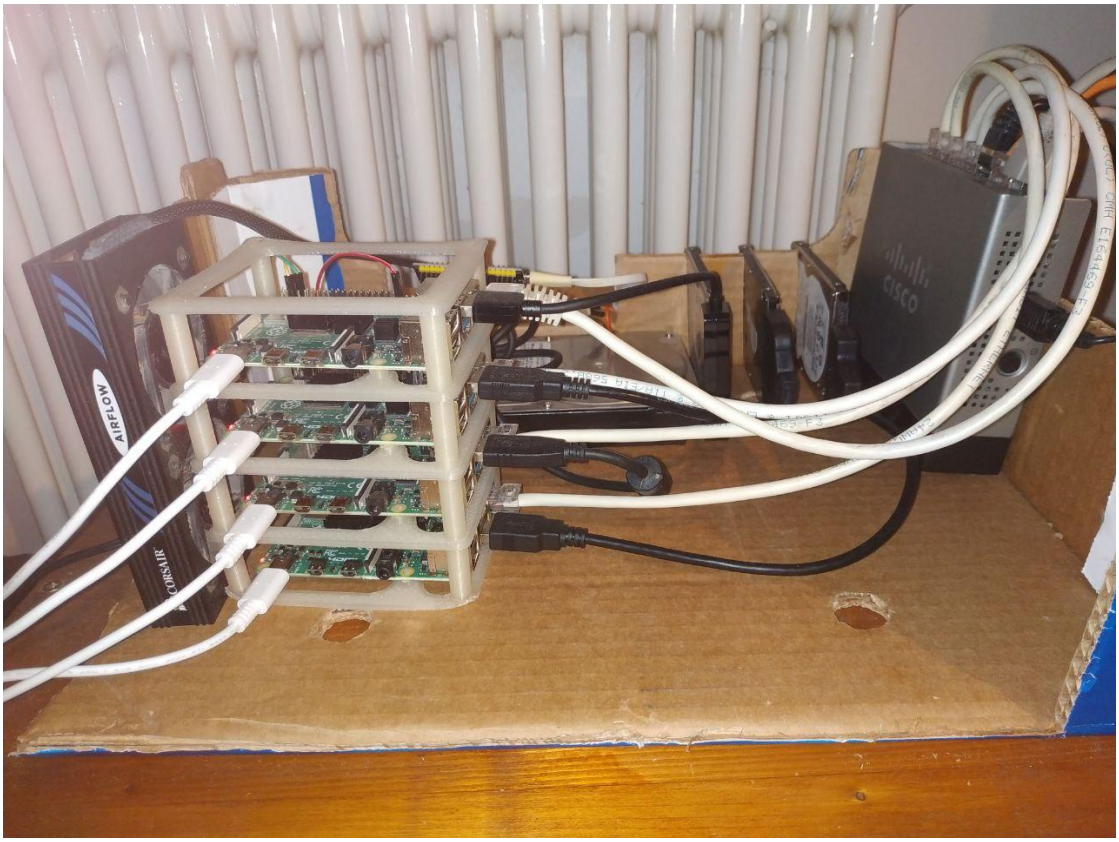


Figura 6.1: Cluster completamente assemblato.

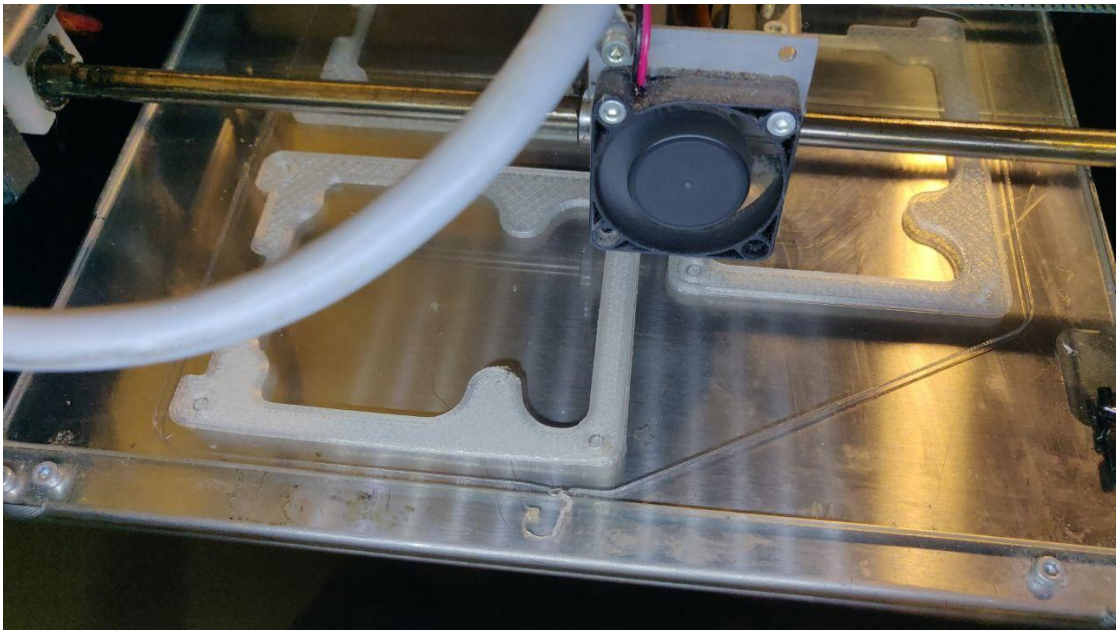


Figura 6.2: Stampa del case.

6.2.1 Installazione BMC

Per realizzare il codice relativo al BMC su base ESP32 è stato utilizzato micropython al contrario del linguaggio C++ usato tipicamente, in quanto, dato che sarebbe stato necessario ospitare un webserver, la realizzazione sarebbe stata troppo complessa.

Il webserver implementa quindi le API descritte in fase di progettazione, ed in base alla richiesta ricevuta va ad operare sul pin corrispondente sui Raspberry, in Tabella 6.1 è riassunto il collegamento tra ESP32 e i vari Raspberry. In particolare, per poter simulare un pulsante è stato necessario impostare sull'ESP i pin collegati sul GPIO3 dei Raspberry in modalità open-drain, che prevede due stati: nello stato logico basso il pin è impostato a massa, mentre nello stato logico alto viene abilitata una resistenza ada alta impedenza, rendendolo scollegato.

In questo modo è quindi possibile simulare dei pulsanti collegati ai pin GPIO dei Raspberry.

MAAS Power Driver

Per permettere l'utilizzo è stato realizzato un Power Driver, da utilizzare all'interno di MAAS per interfacciarsi con il BMC. Per fare ciò si è estesa la classe `WebHookPowerDriver` e sono stati implementati i tre metodi richiesti:

Tabella 6.1: Collegamenti GPIO tra Raspberry ed ESP32.

Raspberry	*GPIO Raspberry*	*GPIO ESP32*
0	14	12
	3	13
1	14	27
	3	26
2	14	33
	3	25
3	14	23
	3	22

power_query Si occupa di verificare lo stato di alimentazione del device gestito, dato l'id. Deve contattare l'endpoint GET /rpis/⟨id⟩ sul BMC;

power_on Si occupa dell'accensione del device, dato l'id, contattando l'endpoint PUT /rpis/⟨id⟩ fornendo come valore di power "on";

power_off Spegne il device desiderato, contattando lo stesso endpoitn di power_on ma con valore off.

Sono state inoltre definite le impostazioni che il power driver dovrà richiedere all'utente per essere configurato nella GUI di MAAS.

6.2.2 Configurazione Proxmox

Per poter utilizzare il provider Proxmox con Terraform è necessario prima eseguire alcuni step preliminari, descritti di seguito.

Creazione utente Terraform in Proxmox

Al fine di poter utilizzare l'API di Proxmox, è necessario creare un utente apposito attraverso la CLI di Proxmox. Con il comando `pveum user add terraform-prov@pve --password $PASSWORD` viene creato l'utente e gli vengono associati i permessi corretti attraverso `pveum aclmod / -user terraform-prov@pve -role Administrator`. È ora possibile utilizzare il provider con Terraform, configurandolo come in Listing 6.1, fornendo la password come variabile d'ambiente.

Listato 6.1: Configurazione provider Proxmox

```
provider "proxmox" {
  pm_api_url      = "https://${var.pve_host["ip"]}:8006/api2/
     json"
  pm_user         = var.pve_host["pm_user"]
  pm_tls_insecure = true
}
```

Listato 6.2: Creazione template Proxmox

```
qm create 9000 -name ubuntu-cloudinit -memory 1024 -net0
    virtio,bridge=vmbr0 -cores 1 -sockets 1
qm importdisk 9000 focal-server-cloudimg-amd64.img local
   -lvm
qm set 9000 -scsihw virtio-scsi-pci -virtio0 local-lvm:
   vm-9000-disk-0
qm set 9000 -boot c -bootdisk virtio0
qm template 9000
```

Creazione Template

Per permettere a Terraform di istanziare VM è necessario creare un un template di macchina virtuale all'interno di Proxmox. Come primo passo è necessario scaricare l'immagine con supporto cloud-init di Ubuntu con il seguente comando.

```
wget https://cloud-images.ubuntu.com/focal/current/focal
   -server-cloudimg-amd64.img
```

Sarà poi possibile creare il template seguendo il procedimento specificato in Listing 6.2: per prima cosa è necessario creare una macchina virtuale nuova, poi può essere importata l'immagine nello storage per poi essere collegata alla macchina virtuale come disco di boot (linee 3 e 4) ed infine è possibile traformarla in template.

6.2.3 Deployment e configurazione automatica VM

Ora è possibile creare VM utilizzando il provider Proxmox per Terraform.

In Listing 6.3 viene mostrata la configurazione per creazione della macchina virtuale su Proxmox attraverso Terraform. In particolare nelle linee 11–22 vengono

specificate e configurate le interfacce di rete con i relativi indirizzi IP, dichiarati nel file di variabili.

Dopo aver eseguito il comando `terraform init` per scaricare le dipendenze ed inizializzare l'ambiente, è quindi possibile applicare la configurazione attraverso i comandi `terraform plan -out plan` e `terraform apply plan`.

Al termine della creazione della macchina virtuale, essa viene configurata eseguendo Ansible in un provisioner locale, come mostrato in Listing 6.4.

6.2.4 Installazione e configurazione MAAS

La parte di installazione e configurazione del software sulla macchina appena viene gestita da un playbook Ansible, eseguito attraverso Terraform. Oltre alla configurazione delle reti, viene applicata una configurazione specifica relativa alla gestione dei Raspberry Pi, mostrata in Listing 6.5.

Per prima cosa viene impostato lo snippet DHCP, sotto forma di template Jinja2 [6.6] per permettere la sostituzione dei parametri di configurazione. Esso permette di effettuare il chainloading del firmware UEFI, usando MAAS come server TFTP. Il firmware UEFI è stato posizionato all'interno della cartella esposta da MAAS tramite TFTP [linea 5]. Per poter utilizzare il power driver è necessario copiarlo all'interno della cartella giusta, in modo da farlo caricare a MAAS. Dato che MAAS è distribuito come snap, è necessario montare la directory con i file necessari sopra a quella all'interno dello snap [linea 34], dopo aver copiato e modificato il file di registry [linee 12–26].

Configurazione IP forwarding

Per permettere agli host presenti sulla rete isolata di accedere ad Internet, si è deciso di utilizzare la macchina MAAS come router tra la prima rete e quella principale. Questo è possibile impostando delle regole con iptables, mostrate in Listing 6.7, che effettuano il NAT e il forwarding fra le due interfacce [8].

Lo script ed è stato trasferito sulla macchina ed impostato per l'esecuzione automatica all'avvio con un cronjob attraverso Ansible.

6.2.5 Registrazione e deployment degli host

Terminata l'installazione automatica della macchina di MAAS, realizzata attraverso la combinazione di Ansible e Terrafom, è stato possibile iniziare a gestire gli host che andranno a comporre il cluster.

Listato 6.3: Creazione macchina virtuale per MAAS attraverso Terraform

```
resource "proxmox_vm_qemu" "vm-01" {

  name        = "maas-tf"
  target_node = var.pve_host["target_node"]

  clone   = "ubuntu-20.04-cloudimg"
  os_type = "cloud-init"

  cores = 2

  ipconfig0 = "ip=${var.maas_host["pub_ip"]}/24,gw=${var.
      maas_host["pub_gw"]}"
  ipconfig1 = "ip=${var.maas_host["priv_ip"]}/24"

  network {
    model  = "virtio"
    bridge = "vmbr0"
  }

  network {
    model  = "virtio"
    bridge = "vmbr1"
  }

  scsihw = "virtio-scsi-pci"

  disk {
    size     = "30G"
    type     = "scsi"
    storage  = "local-lvm"
    iothread = 1
  }
}
```

Listato 6.4: Esecuzione playbook Ansible attraverso Terraform

```
provisioner "local-exec" {
    working_dir = "ansible/"
    command     = "ansible-playbook -u ${var.maas_host["
        maas_user"]} --key-file ${var.ssh_keys["priv"]} -i ${var
        .maas_host["pub_ip"]}, site.yml"
}
```

Creazione VM Kolla controller

Per poter effettuare il deploy di OpenStack attraverso Kolla-Ansible è necessario disporre di un host che svolga la funzione di controller. Per questo scopo si è deciso di utilizzare una macchina virtuale hostata su Proxmox e gestita da MAAS.

È stata quindi creata attraverso la GUI di Proxmox una VM non inizializzata, collegata alla rete gestita da MAAS, impostando la scheda di rete come primo dispositivo di boot.

All'avvio la macchina è stata correttamente riconosciuta ed enlistata e, dopo aver configurato la connessione con il power driver di Proxmox, è stato possibile eseguirne il deploy in maniera automatica.

Raspberry Pi

In modo da effettuare l'operazione di enlist sono stati accesi manualmente i device, in modo da essere rilevati da MAAS.

In seguito a questa prima operazione è stato impostato il power driver, attraverso il quale, dopo avere inserito i parametri necessari per interfacciarsi con il BMC installato in precedenza, è possibile gestire in maniera automatica le fasi di commissioning e di deployment dei Raspberry, come mostrato in Figura 6.3.

Configurazione storage Per permettere l'utilizzo di volumi LVM da parte di Cinder, è stato necessario impostare la configurazione di storage all'interno di MAAS. È stato necessario creare tre partizioni: una riservata al boot, di circa 500MB, una per il file system di root, di circa 250GB ed una per LVM, sempre di 250GB. La terza partizione è poi stata impostata come volume LVM. La configurazione finale è mostrata in Figura 6.4.

Assegnamento tag A deploy completato si è proceduto ad assegnare i tag attraverso MAAS, per permettere di distinguere le macchine nelle fasi successive. In particolare alla macchina virtuale per Kolla si è assegnato il tag *ansible-ctrl*,

Listato 6.5: Configurazione MAAS per la gestione di Raspberry Pi

```
- name: set DHCP snippet
  command: "maas {{ maas_admin }} dhcpsnippets create name='
     raspberry' 'value={{ dhcpd_snippet }}' description='
     raspberry pxe' subnet={{ sub_id }}"
  changed_when: false

- name: Copy UEFI firmware to host
  become: yes
  copy:
    mode: 0775
    src: TFTP_Server/tftpboot/.
    dest: /home/ubuntu-maas/tftpboot

- name: Copy power dir
  copy:
    mode: 0775
    src: /snap/maas/current/lib/python3.8/site-packages/
       provisioningserver/drivers/power/.
    dest: ~/power
    remote_src: yes

- name: Edit registry file
  ansible.builtin.lineinfile:
    path: ~/power/registry.py
    insertafter: '{{item.After}}'
    line: '{{item.Line}}'
  with_items:
    - { After: "power_drivers = [", Line: "RPIPowerDriver(),"}
    - { After: "from provisioningserver.utils.registry import
       Registry", Line: "from provisioningserver.drivers.power.
       rpi import RPIPowerDriver"}

- name: Copy BMC power driver
  copy:
    mode: 0775
    src: ESP32_power_driver/power/.
    dest: ~/power

- name: Mount bmc power driver
  mount:
    path: /snap/maas/current/lib/python3.8/site-packages/
       provisioningserver/drivers/power
    src: /home/ubuntu-maas/power
    fstype: ext4
    opts: bind,nodev,ro
    state: mounted
  become: yes
```

Listato 6.6: Template Jinja2 dello snippet DHCP

```
if option vendor-class-identifier = "PXEClient:Arch:00000:UNDI
    :002001" {
     option vendor-encapsulated-options "Raspberry Pi Boot";
     option tftp-server-name "{{ subnet2_maas_ip }}";
}
```

Listato 6.7: Regole Iptables

```
iptables-restore <<-EOF
*nat
-A POSTROUTING -o "$EXTIF" -j MASQUERADE
COMMIT
*filter
:INPUT ACCEPT [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]
-A FORWARD -i "$EXTIF" -o "$INTIF" -m conntrack --ctstate
    ESTABLISHED,RELATED -j ACCEPT
-A FORWARD -i "$INTIF" -o "$EXTIF" -j ACCEPT
-A FORWARD -j LOG
COMMIT
EOF
```

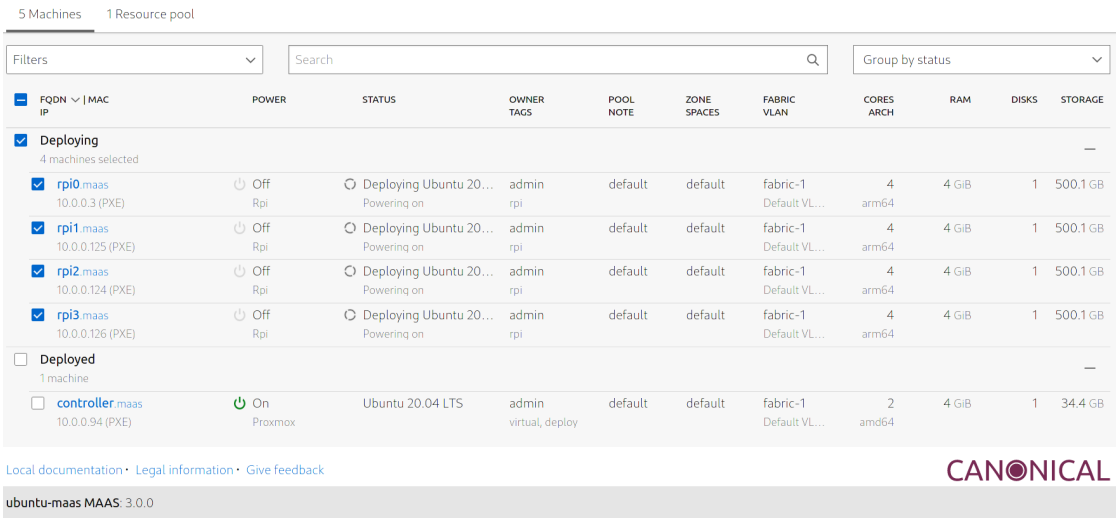
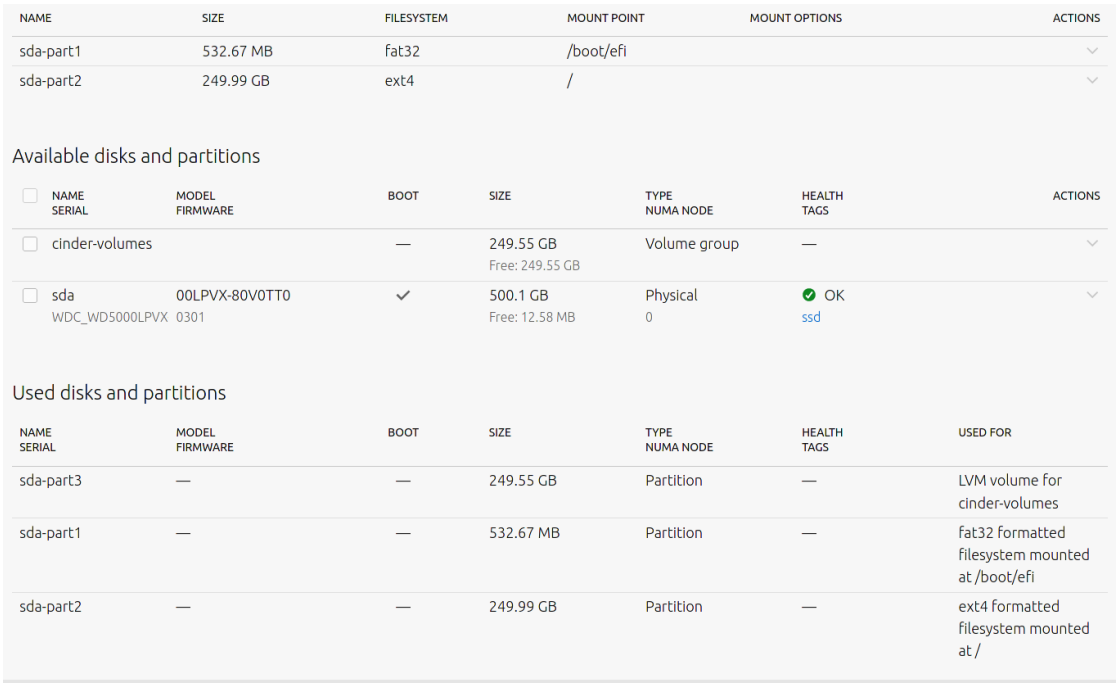


Figura 6.3: Schermata dell'interfaccia web di MAAS durante la fase di deploy.



NAME	SIZE	FILESYSTEM	MOUNT POINT	MOUNT OPTIONS	ACTIONS
sda-part1	532.67 MB	fat32	/boot/efi		
sda-part2	249.99 GB	ext4	/		

Available disks and partitions

NAME SERIAL	MODEL FIRMWARE	BOOT	SIZE	TYPE NUMA NODE	HEALTH TAGS	ACTIONS
cinder-volumes		—	249.55 GB Free: 249.55 GB	Volume group	—	
sda WDC_WD5000LPVX	00LPVX-80V0TT0 0301	✓	500.1 GB Free: 12.58 MB	Physical 0	OK ssd	

Used disks and partitions

NAME SERIAL	MODEL FIRMWARE	BOOT	SIZE	TYPE NUMA NODE	HEALTH TAGS	USED FOR
sda-part3	—	—	249.55 GB	Partition	—	LVM volume for cinder-volumes
sda-part1	—	—	532.67 MB	Partition	—	fat32 formatted filesystem mounted at /boot/efi
sda-part2	—	—	249.99 GB	Partition	—	ext4 formatted filesystem mounted at /

Figura 6.4: Configurazione storage in MAAS.

mentre i Raspberry sono stati riuniti in unico gruppo *rpi*, suddiviso internamente tra *compute* (assegnato ai 3 Raspberry da 4GB di RAM) e *control*, con come unico membro l'SBC da 8GB di RAM.

Problemi riscontrati

Durante il deployment dei Raspberry è stato riscontrato un bug intermittente ancora non risolto [6], che impedisce di portare a termine con successo il deploy di Ubuntu 20.04 LTS.

Consiste nell'errata configurazione della DataSource di cloud-init durante il deploy, utilizzato da MAAS per ricevere gli eventi di cambiamento di stato dalle macchine gestite. Questa configurazione dovrebbe essere gestita interamente da MAAS, ma per una ragione ancora non identificata non viene effettuata la totalità delle volte. Questo impedisce la ricezione dell'evento di fine boot al termine del riavvio eseguito al termine dell'installazione del sistema operativo, per cui la macchina rimane nello stato *Rebooting* fino allo scadere del timeout, causando il fallimento del deploy.

Per ovviare a questa problematica, nei casi in cui si è presentato il problema è stata installata invece la versione 20.10 di Ubuntu, in quanto versione più simile

alla 20.04 LTS. Questo cambiamento ha però richiesto la rimozione di un controllo durante la fase di installazione di OpenStack, descritto in Sezione 6.3.1.

6.3 Deployment Openstack

Una volta effettuato il deploy del sistema operativo sugli host attraverso MAAS, è possibile iniziare la fase di deploy di OpenStack, non prima aver effettuato qualche operazione di configurazione preliminare.

6.3.1 Configurazione preliminare nodi

Prima di poter procedere al deploy è necessario effettuare alcune configurazioni specifiche necessarie a preparare gli host per l'installazione di OpenStack.

Tutte le seguenti configurazioni sono state applicate utilizzando Ansible, che richiede in input, oltre alle azioni da compiere, un *inventory* di host da configurare, che a livello basilare deve essere composto da un elenco di IP o FQDN identificanti gli host, suddivisi per gruppo.

Dato che all'interno di MAAS gli host non hanno indirizzi IP fissi e possono variare nel tempo a causa di nuovi deploy o aggiunta di host, è necessario generare dinamicamente l'inventory. Questo è reso possibile da *ansible maas inventory* [38], con cui è possibile ottenere da MAAS la lista di host controllati, suddivisi in gruppi in base ai tag assegnati precedentemente all'interno del sistema.

È quindi stato possibile far eseguire il playbook da Ansible attraverso il comando `ansible-playbook site.yml -i ansible-maas-dynamic-inventory/maas.py`.

Configurazione di rete È necessario creare due interfacce di rete virtuali (veth), necessarie per OpenStack, di cui una verrà utilizzata attraverso un bridge con l'interfaccia fisica, per consentire la comunicazione con l'esterno. La seconda interfaccia virtuale verrà utilizzata dal servizio di networking SDN di Openstack, Neutron.

Dopo la loro creazione, le interfacce sono state configurate attraverso Ansible, tramite l'utilizzo di ruolo creato appositamente per la gestione della configurazione di rete [40]. Esso opera richiedendo in input un template di configurazione netplan, adattabile attraverso la sostituzione delle variabili tra doppie graffe effettuata dal motore di templating Jinja2, come mostrato in Listing 6.8.

Configurazione archiviazione In modo da poter utilizzare il servizio di storage è necessario fornire un back-end. Avendo deciso di utilizzare LVM, è stato necessario creare un nuovo *volume group* attraverso il comando `vgcreate cinder-volumes /dev/sda3`.

Listato 6.8: Configurazione interfacce virtuali.

```
- name: Setup bridge configuration
  include_role:
    name: mrlesmithjr.netplan
  vars:
    become: yes
    netplan_enabled: true
    netplan_config_file: /etc/netplan/50-cloud-init.yaml
    netplan_renderer: networkd
    netplan_configuration:
      '{
            "network": {
              "ethernets": {
                "{{ physical_interfaces_cmd.stdout }}": {
                  "dhcp4": false,

                },
                "{{ veth1 }}": {},
                "{{ veth2 }}": {}
            },
            "bridges": {
              "{{ br0 }}":{
                "addresses": [
                      "{{ ansible_ssh_host }}/24"
                  ],
                  "gateway4": "{{ maas_host }}",
                  "nameservers": {
                    "addresses": [
                        "{{ maas_host }}"
                    ]
                  },
                  "interfaces": [
                    "{{ physical_interfaces_cmd.stdout }}",
                    "veth1"
                  ]
              }
            }
          }
      }'
```

Listato 6.9: Template utilizzato per la generazione dell'inventory

```
{% for host in groups.rpi %}
{{ host }} ansible_private_key_file=~/.ssh/id_rsa ansible_user=
    ubuntu network_interface=br0 neutron_external_interface=
    veth2 ansible_python_interpreter=/usr/bin/python3
{% endfor %}
[control]

{{ groups.control | first}}

[network]
{% for host in groups.rpi %}
{{ host }}
{% endfor %}

[compute]
{% for host in groups.compute %}
{{ host }}
{% endfor %}

[monitoring]
{{ groups.control | first}}

[storage]
{% for host in groups.compute %}
{{ host }}
{% endfor %}
```

Configurazione controller La configurazione del controller consiste principalmente nell'installazione di Kolla-Ansible e le sue dipendenze, oltre alla copia delle chiavi SSH inserite all'interno di MAAS, in modo da poter accedere ai dispositivi configurati da esso. In seguito viene generato il file di inventory per Kolla-Ansible, recuperando le informazioni sugli host dall'inventory dinamico, per poi inserirli nei gruppi corretti all'interno del template, come mostrato in Listing 6.9, rispecchiando l'architettura descritta in Sezione 5.4.1.

Una volta generato il file di inventory è possibile testarne la correttezza contattando tutti gli host specificati con il comando `ansible -i multinode all -m ping`.

Configurazione Kolla-Ansible La configurazione principale di Kolla-Ansible viene gestita attraverso il file `globals.yml` che per l'appunto gestisce le opzioni di configurazione globali, che coprono la maggior parte dei parametri comunemente modficati. Ulteriori configurazioni specifiche relative al singolo servizio non coperte dal suddetto file sono gestire attraverso file aderenti allo schema di configurazione tradizionale di OpenStack.

Per consentire il corretto deploy sui Raspberry è necessario aggiungere l'opzione `openstack_tag_suffix:"-aarch64"` all'interno di `globals.yml`, in modo da costringere a Kolla-Ansible di scaricare le immagini Docker compilate per ARM, identificate sul registry in base al tag. Inoltre, visto che vengono utilizzati i servizi di HA, è stato necessario specificare un IP virtuale, attraverso il quale è possibile contattare il cluster dall'esterno.

Il file contente le password che verranno utilizzate dai servizi del cluster viene generato attraverso il comando `kolla-genpwd`.

Per ovviare alla problematica descritta in Sezione 6.2.5, è stato necessaria una modifica al codice sorgente di Kolla-Ansible, al fine di aggiungere la versione 20.10 di Ubuntu tra le versioni supportate.

Questo è stato effettuato modificato il file `kolla-ansible/ansible/roles/prechecks/vars/main.yml`, aggiungendo `"groovy"` nella lista di versioni di Ubuntu ammesse.

6.3.2 Deployment OpenStack

Al termine della configurazione degli host è possibile dare il via alle operazioni di deploy di OpenStack. Per fare ciò è necessario accedere via SSH all'host di controllo, e da esso eseguire una sequenza di comandi di Kolla-Ansible, secondo il pattern `kolla-ansible -i multinode <comando>`:

- Il primo comando eseguito è `bootstrap-servers`, attraverso cui verranno installate le dipendenze necessarie;
- In seguito con il comando `prechecks` viene controllato che il sistema sia pronto ad accettare il deploy e che non ci siano problematiche;
- Infine viene eseguito il deploy vero e proprio, con `deploy`, che si occupa di scaricare e istanziare i container e configurarli fra loro al fine di ottenere il cluster.

Una volta terminato il deploy attraverso il comando `post-deploy` è possibile generare il file adminrc, che contiene le credenziali e gli URL degli endpoint, necessari per interagire con il cluster.

Listato 6.10: Output del comando `openstack service list`.

```
+----------------------------------+-------------+----------------+
| ID                               | Name        | Type           |
+----------------------------------+-------------+----------------+
| 2fe797c44f374c64b5f430da3f171cfb | keystone    | identity       |
| 6847caae7e62479781177c9551eb6a89 | heat-cfn    | cloudformation |
| a56e00a82fd94ea1be31b875d6d6624b | heat        | orchestration  |
| ad0e23dd0a814991a3ba053311dbb7aa | placement   | placement      |
| e058caebc7854d63884d5c409b1bfe68 | glance      | image          |
| e1fdbd7079694882a2d81fdee4409f19 | cinderv3    | volumev3       |
| e41d5dce19b5470da55f5e5ceb0527a4 | neutron     | network        |
| f72c0025c52745b4a56f7986e7196422 | nova_legacy | compute_legacy |
| ffab01458e1e4ecb97e78dd9c3b75268 | nova        | compute        |
+----------------------------------+-------------+----------------+
```

6.3.3 Verifica funzionamento base OpenStack

Al termine del deploy per prima cosa si è verificato il funzionamento dei vari servizi.

Interazione via riga di comando

La CLI OpenStack è installabile sul proprio PC attraverso il comando `pip3 install python-openstackclient`. Dopo aver aggiunto all'ambiente della shell corrente le credenziali e gli endpoint attraverso `source adminrc.sh`, è possibile interagire con il cloud.

Attraverso la CLI è possibile innanzitutto verificare l'installazione e il funzionamento dei servizi specificati in fase di progettazione. Una lista globale dei servizi è ottenibile con il comando `openstack service list`, il cui output, mostrato in Listing 6.10, conferma la corretta installazione di tutti i componenti desiderati. Per una visione più dettagliata è possibile utilizzare il comando `openstack endpoint list`, che produce in output la lista di tutti gli endpoint delle API messe a disposizione da OpenStack, incluse quelle utilizzate internamente [Listing 6.11]. Si nota che l'indirizzo IP per tutti gli endpoint è lo stesso, ed è l'indirizzo virtuale assegnato ai servizi di loadbalancing.

Lo stato dei singoli servizi è verificabile in maniera analoga. Attraverso il comando `openstack compute service list` si ottiene lo stato dei servizi di Nova e la divisione fra i vari host [Listing 6.12], mentre con `openstack volume service list` si ottiene lo stato di Cinder [Listing 6.13], dove il servizio di backup risulta down, in quanto il servizio Swift non è stato installato, in quanto non necessario in questa installazione.

Listato 6.11: Output del comando `openstack endpoint list -c 'Service Name' -c Interface -c Enabled -c URL --sort-column Interface` .

```
+--------------+---------+-----------+-----------------------------------------+
| Service Name | Enabled | Interface | URL                                     |
+--------------+---------+-----------+-----------------------------------------+
| placement    | True    | admin     | http://10.0.0.10:8780                   |
| neutron      | True    | admin     | http://10.0.0.10:9696                   |
| cinderv3     | True    | admin     | http://10.0.0.10:8776/v3/%(tenant_id)s  |
| heat         | True    | admin     | http://10.0.0.10:8004/v1/%(tenant_id)s  |
| nova_legacy  | True    | admin     | http://10.0.0.10:8774/v2/%(tenant_id)s  |
| heat-cfn     | True    | admin     | http://10.0.0.10:8000/v1                |
| keystone     | True    | admin     | http://10.0.0.10:35357                  |
| nova         | True    | admin     | http://10.0.0.10:8774/v2.1              |
| glance       | True    | admin     | http://10.0.0.10:9292                   |
| nova_legacy  | True    | internal  | http://10.0.0.10:8774/v2/%(tenant_id)s  |
| neutron      | True    | internal  | http://10.0.0.10:9696                   |
| glance       | True    | internal  | http://10.0.0.10:9292                   |
| placement    | True    | internal  | http://10.0.0.10:8780                   |
| heat         | True    | internal  | http://10.0.0.10:8004/v1/%(tenant_id)s  |
| nova         | True    | internal  | http://10.0.0.10:8774/v2.1              |
| heat-cfn     | True    | internal  | http://10.0.0.10:8000/v1                |
| cinderv3     | True    | internal  | http://10.0.0.10:8776/v3/%(tenant_id)s  |
| keystone     | True    | internal  | http://10.0.0.10:5000                   |
| glance       | True    | public    | http://10.0.0.10:9292                   |
| heat         | True    | public    | http://10.0.0.10:8004/v1/%(tenant_id)s  |
| cinderv3     | True    | public    | http://10.0.0.10:8776/v3/%(tenant_id)s  |
| nova         | True    | public    | http://10.0.0.10:8774/v2.1              |
| keystone     | True    | public    | http://10.0.0.10:5000                   |
| neutron      | True    | public    | http://10.0.0.10:9696                   |
| placement    | True    | public    | http://10.0.0.10:8780                   |
| heat-cfn     | True    | public    | http://10.0.0.10:8000/v1                |
| nova_legacy  | True    | public    | http://10.0.0.10:8774/v2/%(tenant_id)s  |
+--------------+---------+-----------+-----------------------------------------+
```

Listato 6.12: Output del comando `openstack compute service list -c Binary -c Host -c Zone -c Status -c State`.

```
+----------------+------+----------+---------+-------+
| Binary         | Host | Zone     | Status  | State |
+----------------+------+----------+---------+-------+
| nova-scheduler | rpi0 | internal | enabled | up    |
| nova-conductor | rpi0 | internal | enabled | up    |
| nova-compute   | rpi2 | nova     | enabled | up    |
| nova-compute   | rpi1 | nova     | enabled | up    |
| nova-compute   | rpi3 | nova     | enabled | up    |
+----------------+------+----------+---------+-------+
```

Listato 6.13: Output del comando `openstack volume service list -c Binary -c Host -c Zone -c Status -c State`.

```
+------------------+-------------+------+---------+-------+
| Binary           | Host        | Zone | Status  | State |
+------------------+-------------+------+---------+-------+
| cinder-scheduler | rpi0        | nova | enabled | up    |
| cinder-volume    | rpi2@lvm-1  | nova | enabled | up    |
| cinder-volume    | rpi1@lvm-1  | nova | enabled | up    |
| cinder-volume    | rpi3@lvm-1  | nova | enabled | up    |
| cinder-backup    | rpi2        | nova | enabled | down  |
| cinder-backup    | rpi1        | nova | enabled | down  |
| cinder-backup    | rpi3        | nova | enabled | down  |
+------------------+-------------+------+---------+-------+
```

Operazioni preliminari sul cluster

Una volta verificato il funzionamento si è proceduto con le operazioni preliminari sul cluster, per le quali Kolla-Ansible fornisce uno script di inizializzazione apposito.

Eseguendo questo script vengono configurati gli aspetti necessari a fornire un ambiente utilizzabile dall'utente finale: vengono create una rete esterna ed una interna, connesse fra loro con un router, caricata una immagine di test, determinati i tipi di istanze creabili, decise le quote di utilizzo dei servizi, creato il gruppo di sicurezza di default e inserite la chiave SSH per l'utente.

Creazione VM di test

Al termine della configurazione è possibile verificare la possibilità di utilizzare l'ambiente creando una VM di test, utilizzando l'immagine di CirrOS caricata dallo script di inizializzazione. Si esegue quindi il comando

```
openstack server create \\
--image cirros \\
--flavor m1.tiny \\
--key-name mykey \\
--network int-net \\
demo1
```

per creare un'istanza di test. In seguito è possibile osservare la corretta creazione attraverso il comando `openstack server list`. Terminata la creazione, per poter accedere all'istanza dall'esterno è necessario assegnarle un floating IP connesso con la rete esterna. Per allocare l'indirizzo è sufficiente eseguire il comando `openstack floating ip create --project admin --subnet public-subnet public`, che

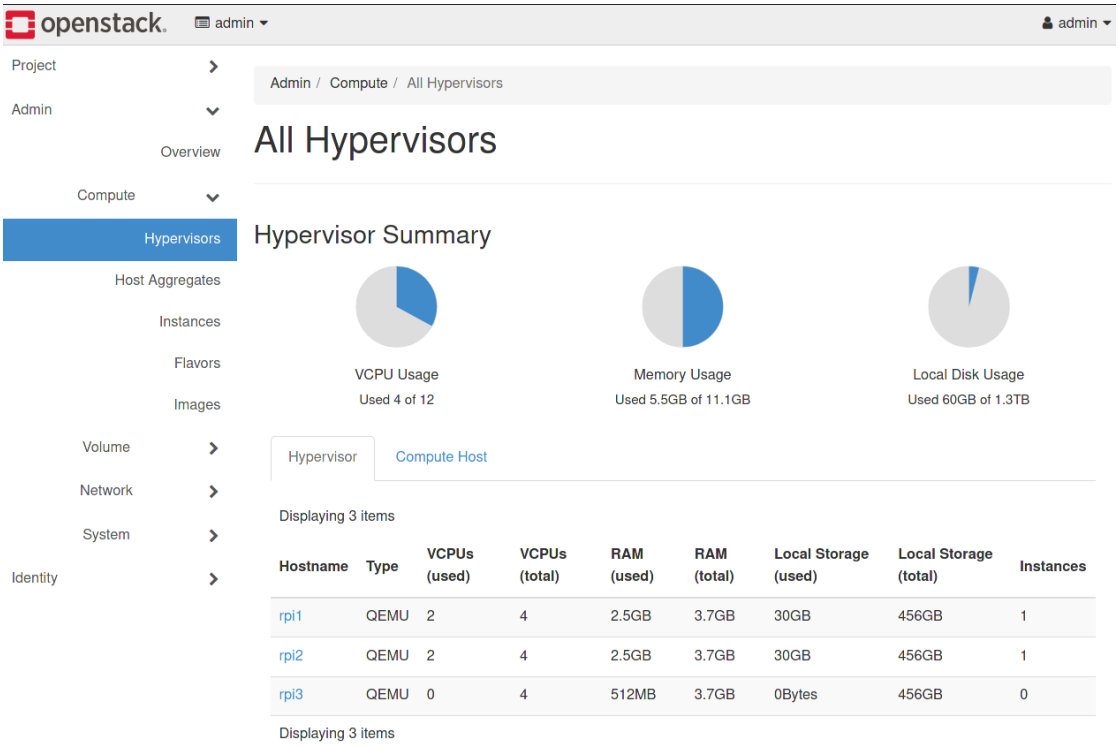


Figura 6.5: Sommario dell'utilizzo degli hypervisor in Horizon.

ad esecuzione completata ritorna l'indirizzo IP allocato. Dopo averlo assegnato con alla macchina con `openstack server add floating ip <istanza> <floating-ip>` è possibile effettuare il login ad essa via SSH utilizzando come indirizzo quello allocato in precedenza e come username e password rispettivamente "cirros" e "gocubsgo". Una volta effettuato l'accesso è possibile verificare il funzionamento della connettività della macchina effettuando un `ping` verso gli host opportuni.

Al termine la macchina può essere deistanziata con il comando `openstack server delete cirros`.

Interazione via Dashboard web

È possibile interagire con il cluster attraverso la dashboard Horizon, accessibile da browser web all'indirizzo specificato come virtual IP durante la configurazione.

Una volta fatto il login con le credenziali dell'utente admin è quindi possibile interagire in maniera grafica con il cluster, oltre a visualizzare varie statistiche di utilizzo. In Figura 6.5 è mostrata la schermata relativa all'utilizzo corrente della capacità di computazione messa a disposizione dagli hypervisor gestiti da OpenStack.

6.4 Deployment Kubernetes

Completata la costruzione e configurazione del cloud privato basato su OpenStack è stato possibile sfruttare le feature di IaaS fornite da esso per la realizzazione del cluster Kubernetes.

6.4.1 k3s

Come base delle VM è stata utilizzato Ubuntu 20.04 LTS, disponibile all'interno del cloud in seguito al caricamento dell'immagine tramite il comando `openstack image create`.

Il numero massimo di nodi è dipendente dal flavor selezionato per gli stessi, mostrati in Listing 6.14. Sulla base di questo, considerando che è necessario utilizzare per il nodo master un flavor di almeno *m1.medium* è possibile identificare due estremi riguardo al numero di macchine possibili all'interno del cluster.

Nonostante sia possibile effettuare overcommitting delle risorse, è sconsigliato farlo, in quanto causerebbe un degradamento generale delle performance, quindi le valutazioni sono state effettuate considerando le risorse disponibili senza overcommitting.

Sono stati valutati prima i due estremi: utilizzando il flavor *m1.xlarge* per tutte le VM è possibile istanziare come massimo un master e due nodi worker. Mentre utilizzando *m1.medium* e *m1.tiny* per i worker è possibile un massimo di 10 worker, così i worker però risulterebbero limitati nelle capacità di computazione, utilizzando la quasi totalità delle risorse per il sistema operativo e l'infrastruttura di Kubernetes.

Considerati questi fattori si è scelto di utilizzare le seguenti dimensioni per i nodi: *m1.large* per il master e *m1.medium* per i worker iniziali, utilizzando per lo scaling automatico del cluster worker di flavor *m1.medium*. Una volta disponibile l'immagine, la creazione del cluster Kubernetes è stata gestita attraverso Terraform.

Utilizzando il provider OpenStack attraverso Terraform è stato quindi possibile creare le istanze compute necessarie al cluster. Nel Listing 6.16 è mostrata una versione semplificata delle risorse dichiarate. Di default vengono creati un master ed un worker e attraverso il parametro count è configurabile un numero variabile di worker iniziali. In particolare, visto che l'installazione di k3s è possibile attraverso un one-liner, si è optato di inserire questa operazione in un file di configurazione cloud-init [Listing 6.15], fornito come parametro user-data alla creazione della VM [linea 15].

Per accedere al cluster è stato assegnato un floating IP alla VM master [linea 18–26], e per poter permettere di recuperare la configurazione è stato generato

Listato 6.14: Output del comando `openstack flavor list -c Name -c RAM -c Disk -c VCPUs`.

```
+-----------+------+------+-------+
| Name      |  RAM | Disk | VCPUs |
+-----------+------+------+-------+
| m1.tiny   |  512 |    1 |     1 |
| m1.small  | 1024 |   20 |     1 |
| m1.medium | 1500 |   20 |     1 |
| m1.large  | 2048 |   30 |     2 |
| m1.xlarge | 3072 |   50 |     3 |
+-----------+------+------+-------+
```

Listato 6.15: Configurazione cloud-init per l'installazione di k3s.

```
#cloud-config
runcmd:
- curl -sfL https://get.k3s.io | K3S_KUBECONFIG_MODE="644" ${K3S_OPTS
    } sh -
```

uno script bash preconfigurato per effetttuare la copia del file di configurazione dal nodo master in maniera automatica [linea 28].

Dopo aver creato il piano con `terraform plan -out plan` ed averlo applicato con `terraform apply plan` è stato quindi possibile scaricare e impostare il file di configurazione attraverso lo script apposito ed utilizzare il cluster.

Verifica operatività

Per verificare il funzionamento è necessario scaricare il client kubectl, attraverso il quale è poi possibile verificare l'operatività di Kubernetes. Visualizzando i nodi facenti parti del cluster con `kubectl get nodes`, come mostrato in Listing 6.17, viene confermato il corretto deployment di un nodo master e di un nodo worker.

6.5 Autoscaler

Il cluster autoscaler è realizzato in GO sfruttando il client di Kubernetes [22] e la libreria gophercloud [10], necessaria per interfacciarsi con OpenStack. L'autoscaler opera controllando gli eventi di watch dei pod. Quando si accorge che un pod non può essere schedulato per mancanza di risorse istanzia una nuova VM attra-

Listato 6.16: Codice Terraform per il deploy di k3s.

```
resource "random_password" "k3s_cluster_secret" {
  length  = 48
  special = false
}

resource "openstack_compute_instance_v2" "master" {
  name            = "k3s-master"
  image_name      = var.image_name
  flavor_name     = var.master_flavor
  key_pair        = var.key_pair
  security_groups = [var.security_groups]
  network {
    name = var.network_name
  }
  user_data = templatefile("cloud-init.yml", { K3S_OPTS = "K3S_TOKEN=
    ${random_password.k3s_cluster_secret.result}" })
}

resource "openstack_networking_floatingip_v2" "fip_master" {
  pool    = var.floating_ip_pool
  address = var.master_fip
}

resource "openstack_compute_floatingip_associate_v2" "fip_master" {
  floating_ip = openstack_networking_floatingip_v2.fip_master.address
  instance_id = openstack_compute_instance_v2.master.id
}

resource "local_file" "get_k3s_config" {
  depends_on = [
    openstack_compute_floatingip_associate_v2.fip_master
  ]
  content  = templatefile("${path.module}/getkubeconfig.tpl", {
      master_ip = "${openstack_compute_floatingip_associate_v2.
      fip_master.floating_ip}" })
  filename = "${path.module}/getkubeconfig.sh"
}
```

Listato 6.17: Output del comando `kubectl get nodes`.

```
NAME           STATUS   ROLES                  AGE   VERSION
k3s-worker-1   Ready    <none>                 21d   v1.21.5+k3s2
k3s-master     Ready    control-plane,master   21d   v1.21.5+k3s2
```

Listato 6.18: Creazione di una nuova macchina da aggiungere al cluster.

```
userData = fmt.Sprintf(userData, openstackinit.K3s_token,
    openstackinit.K3s_url)

server, err := servers.Create(client, keypairs.CreateOptsExt{
    CreateOptsBuilder: servers.CreateOpts{
      Name:              GetNodeName(),
      FlavorRef:         flavorID,
      ImageRef:          imageID,
      SecurityGroups:    []string{openstackinit.SecurityGroupName},
      Networks:          []servers.Network{{UUID: openstackinit.
          NetworkUUID}},
      UserData:  []byte(userData),
    },
  KeyName: key},
  ).Extract()
```

Listato 6.19: Configurazione cloud-init per l'aggiunta di un nodo a k3s.

```
#cloud-config
runcmd:
- export K3S_TOKEN=%s
- export K3S_URL=%s
- curl -sfL https://get.k3s.io | sh -
```

verso le API di OpenStack. Quando rileva nodi poco utilizzati, invece procede a deinstanziare la macchina virtuale relativa dal cluster.

Per adattarlo allo specifico caso d'uso è stato necessario introdurre delle modifiche. Questo autoscaler è stato concepito per utilizzare come nuovi nodi degli snapshot di macchine virtuali preconfigurate per il boostrapping dei worker, che devono essere create manualmente dopo la creazione di ogni singolo cluster Kubernetes. Questo aspetto va a minare la semplicità di utilizzo, richiedendo l'esecuzione di questa operazione manuale prima di poter utilizzare l'autoscaler.

Per risolvere questo problema si è aggiunta la possibilità di definire una configurazione cloud-init [Listing 6.19] da passare alle nuove macchine istanziate, configurabile con i comandi per installare k3s ed effettuare il join al cluster.

Quando viene creata una nuova macchina, come mostrato in Listing 6.18, vengono quindi sostituite le configurazioni necessarie all'interno del file cloud-init, e viene passato alla funzione di creazione.

Listato 6.20: Dockerfile per il cluster autoscaler.

```
FROM scratch
ADD bin/autoscaler /
CMD ["/autoscaler"]
```

6.5.1 Docker

Per consentire l'esecuzione all'interno del cluster è necessario creare un container per l'autoscaler.

Per ridurre al minimo le dimensioni del container finale si è deciso di sfruttare la feature di linking statico binari offerta da GO. In questo modo, dopo aver compilato l'eseguibile con `CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o bin/autoscaler cmd/main.go`, è stato possibile creare il container basandosi sull'immagine più minimale possibile offerta da Docker, scratch.

Con tre comandi, come mostrato in Listing 6.20, si è quindi specificato un container con come base un'immagine di dimensione 0. Dopo aver completato la fase di build si è effettuato il push dell'immagine sul registry DockerHub, in modo da poterla scaricare in fase di deploy.

6.5.2 Installazione autoscaler

Per permettere l'esecuzione del pod relativo all'autoscaler è stato creato un deployment, che istanzia un pod, il quale eseguirà l'immagine preparata in precedenza.

Per permettere l'accesso da parte del pod alle risorse necessarie, esposte attraverso le API di Kubernetes, è stato definito un ClusterRole apposito, basandosi su quello usato nel progetto ufficiale dell'autoscaler nativo [24]. Esso è stato poi associato attraverso un ClusterRolebinding ad un ServiceAccount, utilizzato poi dal pod per accedere alle API di Kubernetes.

Per permettere l'accesso alle API di OpenStack, per istanziare e deistanziare VM, le credenziali sono state passate configurando un Secret relativo al pod.

Una volta forniti al cluster i due file contenenti il secret ed il deployment assieme ai relativi role tramite `kubectl apply` è stato possibile verificare il corretto istanziamento attraverso il comando `kubectl get pod -A`, di cui è visibile l'output in Listing 6.21.

Listato 6.21: Output del comando `kubectl get pod -A`.

```
1
2  NAMESPACE     NAME                                      READY   STATUS      RESTARTS   AGE
3  kube-system   helm-install-traefik-crd-hrl5b            0/1     Completed   0          20d
4  kube-system   helm-install-traefik-84plt                0/1     Completed   1          20d
5  kube-system   svclb-traefik-c9vh9                       2/2     Running     2          20d
6  kube-system   coredns-7448499f4d-9zqks                  1/1     Running     1          20d
7  kube-system   traefik-97b44b794-glxpw                   1/1     Running     1          20d
8  kube-system   local-path-provisioner-5ff76fc89d-6fbbt   1/1     Running     1          20d
9  kube-system   metrics-server-86cbb8457f-vbnbc           1/1     Running     1          20d
10 kube-system   svclb-traefik-swfzw                       2/2     Running     2          20d
11 kube-system   autoscaler-6c95f7f5cd-6cz9k               1/1     Running     0          18s
```

Capitolo 7

Validazione dei risultati

In questo capitolo verrà verificato il raggiungimento degli obiettivi posti nel Capitolo 3, descrivendo per ognuno le prove effettuate ed i risultati ottenuti.

7.1 Realizzazione fisica del cluster

La fase di realizzazione fisica del cluster non ha presentato particolari problemi, le macchine selezionate si sono rivelate adatte alle esigenze computazionali. Come descritto in Sezione 4.1 è stato fondamentale l'utilizzo di un Raspberry Pi da 8GB per il nodo controller, mentre i 3 Raspberry da 4GB si sono rivelati adeguatamente performanti per il loro compito.

I dischi utilizzati si sono rivelati sufficientemente performanti, permettendo un deploy del sistema operativo attraverso MAAS in circa 10-15 minuti. È stata considerata anche la possibilità di utilizzare dischi SSD SATA, per ottenere performance ancora migliori, ma poi scartata per le ragioni descritte di seguito. Rispetto ai dischi meccanici, che hanno un consumo di corrente medio di circa 0,5A, con un picco di 0,8A per la fase di spin up dei piatti, un tipico SSD da 2.5" richiede durante la fase di startup un picco di anche 1.7A per caricare i condensatori, con consumi molto più ridotti in seguito.

Questa corrente di picco è un problema per l'utilizzo con un Raspberry Pi, in quanto le sue porte USB supportano un carico massimo di 1.2A. L'utilizzo di un SSD causerebbo quindi problemi nella fase di boot, rendendo necessario l'utilizzo di un'alimentazione esterna, incrementando l'ingombro del cluster.

7.2 Gestione di un bare metal cloud

Dopo le modifiche introdotte per introdurre la compatibilità con i Raspberry, MAAS si è rivelato adatto allo scopo, fornendo un sistema semplice e veloce per

la costruzione di cluster bare metal.

L'aver gestito l'installazione e la configurazione della macchina di MAAS in maniera automatica attraverso tool di IaC come Terraform e Ansible ha incrementato il grado di automazione generale nella costruzione del cluster e garantito una più semplice replicabilità del progetto in futuro.

7.3 Deployment del cluster OpenStack

La costruzione del cluster ha richiesto un considerevole ammontare di tempo, dovuto alla complessità e alla quantità dei servizi da installare sul nodo controller. Mentre le fasi di check preliminari e di bootstrap sono state svolte in circa 10 minuti, la fase di deploy ha richiesto circa un'ora.

Mentre per la fase iniziale di deploy si è scelto di eseguire il deploy eseguendo uno per volta i comandi dall'host di controllo, è stato preparato anche un task Ansible che lancia il procedimento in maniera automatica. Attraverso questo modulo è anche possibile effettuare il join di macchine in un momento successivo alla prima messa in essere del cluster, fornendo in input l'hostname, attraverso il comando riportato di seguito:

```
ansible-playbook site.yml -i ansible-maas-dynamic-
   inventory/maas.py --limit ansible-ctrl:compute
```

In questo modo, in circa 30 minuti, di cui 10 per il deploy attraverso MAAS e il resto per la configurazione di OpenStack, è possibile aggiungere un nuovo nodo compute al cluster.

È stato poi verificato il corretto funzionamento di tutti i servizi installati attraverso l'istanziazione di macchine di prova basate su CirrOS, come descritto anche in Sezione 6.3.3.

In generale il grado di automazione raggiunto nella creazione del cluster OpenStack è stato elevato, rendendo disponibile un ambiente già funzionante in un tempo relativamente breve.

7.4 Deployment del cluster Kubernetes

Attraverso Terraform è stato possibile interagire agilmente con OpenStack e creare le risorse necessarie al cluster Kubernetes. La distribuzione k3s si è rivelata adatta allo scopo, permettendo di creare un cluster con tutte le funzionalità di un cluster di produzione, offrendo semplicità e velocità di installazione e un consumo contenuto di risorse.

Il testing funzionale del cluster è stato effettuato in concomitanza col testing dell'autoscaler, descritto di seguito.

7.5 Scalabilità automatica del cluster Kubernetes

Per verificare la funzionalità del cluster autoscaler sono stati effetuati dei test di carico. Per prima cosa è stato creato un deployment che saturasse le risorse del cluster, impostando l'utilizzo dei di risorse dei container al 100% delle CPU disponibile, riportato in Listing 7.1. Questo è stato ottenuto impostando le risorse richieste ad un intero core di cpu[linea 21], permettendo così di influenzare il grado di utilizzo dei nodi in maniera agevole, facendo scattare il meccanismo di autoscaling.

Inizialmente il numero di repliche è stato impostato a due, occupando così i nodi master e worker, come mostrato in Listing 7.2. In seguito, attraverso il comando `kubectl scale deployment nginx-deployment --replicas 5` si sono incrementate le repliche a 5.

In seguito a questa operazione l'autoscaler ha reagito all'evento di risorse insufficienti, istanziando e configurando una nuova VM, disponibile come nodo nel cluster Kubernetes nel giro di 2-3 minuti. Con il comando `kubectl get pods -o wide` si è potuto verificare l'aggiunta di nuovi nodi e lo scheduling su di essi di un pod, come mostrato in Listing 7.3.

L'effetto contrario è ottenibile riducendo il numero di repliche, in tal caso, al termine del delay preimpostato i nodi inutilizzati sono stati rimossi correttamente.

7.6 Compatibilità del software con l'architettura ARM

In generale la combatibilità con ARM riscontrata è buona, richiedendo solamente in alcuni casi l'inserimento di flag appositi, come per esempio nel caso di Kolla. Sono stati però riscontati casi come quello descritto in Sezione 4.2, dove la compatibilità con arm era quasi totalmente assente, rendendo necessario operazioni non documentate anche solo per provare a far funzionare il servizio. Questo è dovuto alla complessità e alla suddivisione in progetti di OpenStack, che può portare a casi come questo, dove non è stata data tanta importanza alla compatibilità con arm.

Sì può concludere che i software utilizzati sono pienamente compatibili con l'architettura ARM.

Listato 7.1: File di deployment utilizzato per i test.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: arm64v8/nginx:1.20
        resources:
          requests:
            memory: "64Mi"
            cpu: "1000m"
          limits:
            memory: "128Mi"
            cpu: "1000m"
        ports:
        - containerPort: 80
```

Listato 7.2: Output di `kubectl get pods -o wide` in seguito alla creazione del deployment.

```
NAME                                READY   STATUS    RESTARTS   AGE     IP           NODE
nginx-deployment-fbdfb8ccb-5vrq4    1/1     Running   0          12m     10.42.0.8    k3s-master
nginx-deployment-fbdfb8ccb-gpnlz    1/1     Running   0          8m38s   10.42.1.5    k3s-worker-1
```

Listato 7.3: Output di `kubectl get pods -o wide` dopo lo scaling del deployment.

```
NAME                                READY   STATUS    RESTARTS   AGE    IP           NODE
nginx-deployment-fbdfb8ccb-5vrq4    1/1     Running   0          19m    10.42.0.10   k3s-master
nginx-deployment-fbdfb8ccb-gpnlz    1/1     Running   0          19m    10.42.1.6    k3s-worker-1
nginx-deployment-fbdfb8ccb-pd5rb    1/1     Running   0          17m    10.42.2.3    kube-worker-ixzr
nginx-deployment-fbdfb8ccb-6qs59    1/1     Running   0          8m4s   10.42.3.3    kube-worker-65ns
nginx-deployment-fbdfb8ccb-wcxt9    1/1     Running   0          8m4s   10.42.4.3    kube-worker-4u7j
```

Capitolo 8

Conclusioni

In conclusione, è possibile affermare di aver raggiunto tutti gli obiettivi che ci si era posti.

Utilizzando SBC Raspberry Pi e dischi di recupero è stato possibile mantenere il costo del cluster il più basso possibile. Attraverso MAAS ed un BMC realizzato ad hoc è stato possibile gestire in maniera automatica la parte più di basso livello di installazione del sistema operativo.

Utilizzando Kolla-Ansible è stato possibile costruire e gestire il cluster OpenStack in maniera agevole. Il suddetto cluster si è rivelato poi sufficientemente performante, permettendo, attraverso i servizi di IaaS offerti, il deploy di un cluster Kubernetes.

All'interno di questo cluster Kubernetes è stato poi installato un cluster autoscaler, in grado di espandere o contrarre i nodi che compongo il cluster, in base alle risorse richieste dai pod in esecuzione.

Con l'utilizzo di Ansible e Terraform è stato possibile automatizzare le parti tipicamente manuali di creazione di risorse e configurazione delle stesse, rendendo inoltre l'installazione facilmente riproducibile.

8.1 Sviluppi futuri

Nonostante i risultati soddisfino gli obiettivi che erano stati fissati, ci sono degli aspetti e alternative che sarebbe interessante approfondire.

Mentre la soluzione basata su MAAS per la gestione del bare metal si è rivelata adeguata allo scopo, sarebbe interessante utilizzare OpenStack Ironic per questo aspetto. In questo modo tutti gli aspetti di gestione del cloud sarebbero all'interno dell'ambiente OpenStack, permettendo l'utilizzo di progetti come Kayobe [Sezione 2.3.4]. Questo però richiede sicuramente di rivalutare il BMC, in quanto per renderlo compatibile potrebbero essere necessarie corpose modifiche.

Nella fase di deploy attraverso Kolla-Ansible sarebbe interessante valutare il miglioramento di performance offerto da un container registry privato hostato in locale, in alternativa a quello pubblico utilizzato in questo progetto.

Per quanto sia stata utilizzata per garantire l'isolamento del cluster Kubernetes, creando una rete virtuale separata, la funzionalità di SDN di OpenStack è molto potente, sarebbe opportuno esplorarla in maniera più approfondita.

Mentre in questo lavoro ci si è concentrati su quelli principali, sarebbe interessante approffondire tutti i servizi disponilbili all'interno di OpenStack. Inoltre si potrebbe sperimentare con hypervisor alternativi a KVM, come per esempio Xen.

Per quanto i risultati ottenuti con k3s e il relativo autoscaler siano sufficienti, non è stato sfruttata la funzionalità cloud-provider per OpenStack. In futuro si potrebbe implementare una soluzione che implementi questo componente, al fine di garantire una migliore integrazione di Kubernetes con il cloud sottostante.

Bibliografia

[1] Apache. Apache mesos. http://mesos.apache.org/. Accessed: 2021-11-29.

[2] Astronomy. Esp32 model. https://forum.fritzing.org/t/fritzing-part-of-an-esp32/5355/3. Accessed: 2021-11-30.

[3] Karen Bruner. Kubernetes autoscaling - 3 common methods explained. https://cloud.redhat.com/blog/kubernetes-autoscaling-3-common-methods-explained. Accessed: 2021-12-3.

[4] cannikin. Modular raspberry pi cluster case. https://www.thingiverse.com/thing:3743811. Accessed: 2021-12-1.

[5] Canonical. Microstack. https://microstack.run/docs. Accessed: 2021-11-23.

[6] MAAS Discourse. Maas os deployment is failing with cloud init error. https://discourse.maas.io/t/maas-os-deployment-is-failing-with-cloud-init-error/1759/6. Accessed: 2021-12-1.

[7] Docker. Swarm mode overview. https://docs.docker.com/engine/swarm/. Accessed: 2021-11-29.

[8] Ubuntu Documentation. Router. https://help.ubuntu.com/community/Router#Enable_IP_forwarding_and_Masquerading. Accessed: 2021-12-1.

[9] Raspberry Foundation. Raspberry pi 4 tech specs. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. Accessed: 2021-12-1.

[10] gophercloud. Gophercloud: an openstack sdk for go. https://github.com/gophercloud/gophercloud. Accessed: 2021-12-3.

[11] Peter Mell; Timothy Grance. The nist definition of cloud computing. Technical report, National Institute of Standards and Technology: U.S. Department of Commerce., 2011.

[12] HashiCorp. Provisioning infrastructure with terraform. https://www.terraform.io/docs/cli/run/index.html. Accessed: 2021-11-28.

[13] HashiCorp. Terraform language documentation. https://www.terraform.io/docs/language/index.html. Accessed: 2021-11-28.

[14] Red Hat. Learning ansible basics. https://www.redhat.com/en/topics/automation/learning-ansible-tutorial. Accessed: 2021-11-27.

[15] Red Hat. What is a hypervisor? https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor/. Accessed: 2021-11-29.

[16] Red Hat. What is hybrid cloud? https://www.redhat.com/en/topics/cloud-computing/what-is-hybrid-cloud. Accessed: 2021-11-30.

[17] Red Hat. What is infrastructure automation? https://www.redhat.com/en/topics/automation/what-is-infrastructure-automation. Accessed: 2021-11-27.

[18] Red Hat. What is private cloud? https://www.redhat.com/en/topics/cloud-computing/what-is-private-cloud. Accessed: 2021-11-30.

[19] IBM. Terraform. https://www.ibm.com/cloud/learn/terraform. Accessed: 2021-11-28.

[20] iPXE. ipxe - open source boot firmware. https://ipxe.org/. Accessed: 2021-11-27.

[21] juju. Clouds. https://juju.is/docs/olm/clouds. Accessed: 2021-11-26.

[22] Kubernetes. client-go. https://github.com/kubernetes/client-go. Accessed: 2021-12-3.

[23] kubernetes. Cluster autoscaler. https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler#deployment. Accessed: 2021-12-1.

[24] Kubernetes. Cluster autoscaler service account. `https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/magnum/examples/cluster-autoscaler-svcaccount.yaml`. Accessed: 2021-12-3.

[25] kubernetes. Kubernetes components. `https://kubernetes.io/docs/concepts/overview/components/`. Accessed: 2021-11-29.

[26] kubernetes. Namespaces. `https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/`. Accessed: 2021-11-29.

[27] kubernetes. Pods. `https://kubernetes.io/docs/concepts/workloads/pods/`. Accessed: 2021-11-29.

[28] kubernetes. Service. `https://kubernetes.io/docs/concepts/services-networking/service/`. Accessed: 2021-11-29.

[29] kubernetes. Using rbac authorization. `https://kubernetes.io/docs/reference/access-authn-authz/rbac/`. Accessed: 2021-11-29.

[30] kubernetes. What is kubernetes? `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`. Accessed: 2021-11-28.

[31] kubernetes. Workloads. `https://kubernetes.io/docs/concepts/workloads/`. Accessed: 2021-11-29.

[32] Linux. veth(4) — linux manual page. `https://man7.org/linux/man-pages/man4/veth.4.html`. Accessed: 2021-12-1.

[33] Arm ltd. Arm architecture: A foundation for computing everywhere. `https://www.arm.com/why-arm/architecture/cpu`. Accessed: 2021-11-30.

[34] MAAS. About controllers. `https://maas.io/docs/snap/3.0/ui/controllers`. Accessed: 2021-11-27.

[35] MAAS. Maas: how it works. `https://maas.io/how-it-works`. Accessed: 2021-11-26.

[36] Maciej Siczek Maciej Kucia. Deploying openstack - what options do we have? https://www.openstack.org/videos/summits/denver-2019/deploying-openstack-what-options-do-we-have. Accessed: 2021-11-23.

[37] Scott McCarty. A practical introduction to container terminology. https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction#. Accessed: 2021-11-30.

[38] mlimaloureiro. ansible-maas-inventory. https://github.com/mlimaloureiro/ansible-maas-dynamic-inventory. Accessed: 2021-12-1.

[39] Lorenzo Mondani. *Deployment e scaling automatici di un cluster Kubernetes low cost su architettura ARM*. Tesi di laurea magistrale, Alma Mater Studiorum, Università degli Studi di Bologna, 2019/2020.

[40] mrlesmithjr. An ansible role to manage netplan. https://galaxy.ansible.com/mrlesmithjr/netplan. Accessed: 2021-12-1.

[41] OpenStack. Bare metal service overview. https://docs.openstack.org/ironic/latest/install/get_started.html. Accessed: 2021-11-28.

[42] OpenStack. Bare metal service user guide. https://docs.openstack.org/ironic/wallaby/user/index.html. Accessed: 2021-11-27.

[43] OpenStack. Bifrost documentation. https://docs.openstack.org/bifrost/wallaby/. Accessed: 2021-11-26.

[44] OpenStack. Glance documentation. https://docs.openstack.org/glance/wallaby/. Accessed: 2021-11-17.

[45] OpenStack. Haproxy guide. https://docs.openstack.org/kolla-ansible/latest/reference/high-availability/haproxy-guide.html. Accessed: 2021-11-26.

[46] OpenStack. Introduction to the block storage service. https://docs.openstack.org/cinder/wallaby/configuration/block-storage/block-storage-overview.html. Accessed: 2021-11-22.

[47] OpenStack. Keystone architecture. `https://docs.openstack.org/keystone/wallaby/getting-started/architecture.html`. Accessed: 2021-11-20.

[48] OpenStack. Kolla's deployment philosophy. `https://docs.openstack.org/kolla-ansible/wallaby/admin/deployment-philosophy.html`. Accessed: 2021-11-26.

[49] OpenStack. Magnum documentation. `https://docs.openstack.org/magnum/wallaby/`. Accessed: 2021-11-22.

[50] OpenStack. Neutron concepts. `https://docs.openstack.org/neutron/wallaby/install/concepts.html`. Accessed: 2021-11-21.

[51] OpenStack. Nova system architecture. `https://docs.openstack.org/nova/wallaby/user/architecture.html`. Accessed: 2021-11-17.

[52] OpenStack. Supported charms. `https://docs.openstack.org/charm-guide/latest/reference/openstack-charms.html`. Accessed: 2021-11-25.

[53] OpenStack. Swift documentation. `https://docs.openstack.org/swift/wallaby/`. Accessed: 2021-11-22.

[54] OpenStack. Tripleo documentation. `https://docs.openstack.org/tripleo-docs/latest/`. Accessed: 2021-11-26.

[55] Proxmox. Proxmox feautures. `https://www.proxmox.com/en/proxmox-ve/features/`. Accessed: 2021-11-29.

[56] Rancher. K3s - lightweight kubernetes. `https://rancher.com/docs/k3s/latest/en/`. Accessed: 2021-11-29.

[57] Rancher. k3s networking. `https://rancher.com/docs/k3s/latest/en/networking/`. Accessed: 2021-12-3.

[58] Inc. Red Hat. Ansible in depth. Technical report, Red Hat, 2017.

[59] redhat. File storage, object storage o block storage? https://www.redhat.com/it/topics/data-storage/file-block-object-storage. Accessed: 2021-11-22.

[60] Chathura Siriwardhana. Kubernetes cluster autoscaler for openstack. https://chathura-siriwardhana.medium.com/kubernetes-cluster-autoscaler-for-openstack-8effabe5a776. Accessed: 2021-12-1.

[61] sshuttle. sshuttle: where transparent proxy meets vpn meets ssh. https://github.com/sshuttle/sshuttle. Accessed: 2021-11-28.

[62] vanepp. Raspberry pi 4 model. https://forum.fritzing.org/t/raspberry-pi-4-model-b/8622/9. Accessed: 2021-11-30.

[63] Inc. VMware. The architecture of vmware esxi. Technical report, VMware, 2008.