POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

On demand Cloud-based video rendering

Analisi dei costi e delle prestazioni in ambiente Azure

Relatore

prof. Giovanni Malnati

Candidato

Ermete Cinelli

Anno Accademico 2021-2022

Alla mia famiglia

Abstract

I social media hanno cambiato completamente il nostro modo di comunicare. Nel corso di questa rivoluzione, i contenuti multimediali, e nello specifico i video, hanno assunto un ruolo cardinale nelle strategie divulgative. Alla luce di questa analisi, diventa sempre più importante, per le aziende del settore, avere dei meccanismi di rendering efficienti, performanti e sostenibili, al fine di rispondere in maniera adeguata a tale fenomeno, scalando opportunamente e mantenendo inalterate le prestazioni.

Questa esigenza viene analizzata nella presente Tesi di Laurea Magistrale che ha dunque come oggetto la progettazione di un'infrastruttura Cloud based per il rendering on demand di video attraverso Adobe After Effects. Vengono mostrate inizialmente le diverse tecnologie Cloud Native studiate nel corso dell'analisi del problema.

Si arriva poi alla presentazione dell'architettura, alla storia e all'evoluzione della stessa, con i diversi adattamenti dovuti ai limiti della versione Azure di Kubernetes.

Sul piano tecnologico la soluzione prevede l'utilizzo di strumenti come Kubernetes e KubeMQ, tra gli altri, appartenenti al progetto CNCF Cloud Native Landscape. Vengono quindi analizzati i vantaggi che queste soluzioni, pensate per il Cloud, portano rispetto ad altre soluzioni, nate per ambienti non Cloud, che stanno cercando di adattarsi per muoversi in questa direzione.

Altro aspetto fondamentale nella realizzazione di un progetto, ed analizzato nell'elaborato, è quello relativo al monitoring e all'importanza dell'analisi del sistema pensato ed implementato, al fine di capire in quali punti e come intervenire per riuscire a migliorare un'infrastruttura.

Segue infine l'analisi dei costi, con la presentazione di diverse opzioni, tra cui il cliente può scegliere, al fine di avere il giusto compromesso tra le performance e l'esborso di denaro previsto.

Indice

Non est ad astra mollis e terris via

[LUCIO ANNEO SENECA, HERCULES FURENS]

Elenco delle tabelle

Elenco delle immagini

Capitolo Primo

Capitolo 1

1 Dominio applicativo

1.1 Introduzione

Con lo sviluppo dei social, sempre più presenti nelle nostre vite, è completamente mutato il modo in cui comunichiamo. Grazie al Web 2.0, l'utente medio riesce a pubblicare contenuti, e quindi ad esprimersi, in maniera abbastanza semplice, facendo arrivare la sua idea potenzialmente a chiunque sia dotato di un PC e di una connessione ad Internet.

Il quadro perfetto di questa evoluzione nei meccanismi comunicativi può essere identificato nella storia dei Social Network. Tutto inizia da Facebook e simili, i primi social capaci di rendere l'utente in grado di poter esternare le proprie idee attraverso delle frasi, i post. Superato e consolidato Facebook, le successive generazioni sono migrate verso Social come Instagram, i quali hanno come focus principale l'immagine. Si arriva poi ai giorni nostri con la crescita esponenziale di Social quali TikTok che basano i loro contenuti su video.

Da quanto detto sopra si evince come, nel corso del tempo, la comunicazione sul web sia passata dalla classica "frase", a contenuti sempre più di impatto e capaci di colpire e sorprendere l'utente. Seguendo questa tendenza, il mondo di internet, dunque, ha ed avrà sempre di più, sete di immagini e di video.

Per rispondere a tale fenomeno è necessario che, le aziende del settore, dispongano dei giusti strumenti e di una infrastruttura adeguata. Lo scopo finale di tale lavoro di tesi è dunque quello di realizzare un'infrastruttura cloud based capace di renderizzare video e in grado di gestire in maniera autonoma il carico delle richieste in arrivo, scalando in maniera opportuna.

Segue poi un'analisi, tecnica ed economica, dell'architettura concepita, al fine di capire quali siano i reali vantaggi del Cloud e quali siano invece i limiti e le difficoltà a cui si va incontro nel momento in cui si decide di adottare questo tipo di soluzione.

1.2 Sistemi distribuiti

Per poter comprendere appieno quale sia l'ambito e le problematiche affrontate nel corso di questo elaborato, è necessario fissare il concetto di "sistema distribuito".

Con l'espressione sistema distribuito, in informatica, ci si riferisce ad una categoria di sistema informatico costituito da più processi che comunicano attraverso lo scambio di messaggi. La nascita di tali sistemi è dettata sia da motivazioni economiche e sia da motivazioni tecniche.

Tra le motivazioni tecniche emerge sicuramente il fatto che si sia arrivati alla fine di leggi empiriche formulate nello scorso secolo quali la legge di Moore e la legge di Dennard[1]. La conseguenza di ciò si traduce nel fatto che il miglioramento delle performance degli algoritmi non si basa più sull'aumento della frequenza dei processori sui quali girano, ma sulla parallelizzazione degli stessi.

Le difficoltà che devono essere affrontate però, nel momento in cui si decide di realizzare questo tipo di soluzione, non sono affatto banali. In primis possiamo citare la questione relativa al fatto che, in questi sistemi, è presente una grande eterogeneità, essa è intesa sia da un punto di vista hardware, con i numerosi nodi[2] diversi tra loro, e sia dal punto di vista software, ad esempio per quanto riguarda l'utilizzo di diversi linguaggi di programmazione. Altra caratteristica fondamentale dei sistemi distribuiti è la concorrenza; possono essere infatti eseguite contemporaneamente operazioni da due o più macchine differenti; devono dunque essere ideati dei meccanismi, come i lock o i semafori in ambiente non distribuito, capaci di gestire queste situazioni, garantendo, in alcuni casi, accesso esclusivo alle risorse. Deve inoltre essere considerata l'inesistenza di

[1] Le leggi di Moore e di Dennard sono molto collegate tra loro. Si tratta di due leggi empiriche formulate entrambe nella seconda metà del XX secolo, che portano alla conclusione che, grazie al progresso tecnologico, la frequenza dei processori aumenta di circa 2,7 volte con una cadenza biennale.

[2] Macchine sulle quali vengono eseguiti i processi in un sistema distribuito

uno stato generale, la mancanza di un clock globale attraverso il quale potrebbero essere sincronizzati i vari processi in esecuzione su macchine diverse, vengono dunque utilizzate altre tecniche che si basano sull'ordine degli eventi.

Peculiarità primaria di un sistema distribuito è inoltre la possibilità che possano verificarsi malfunzionamenti solo in alcuni nodi, si parla in questo caso di *partial failure*. Tra i requisiti fondamentali di questo tipo di sistemi, vi è la capacità di saper rilevare, risolvere e resistere a tali situazioni di errore senza che queste possano impattare sulle performance e sulla corretta funzionalità del sistema, si parla appunto di *fault tolerance*.

Si può affermare dunque che i sistemi distribuiti garantiscono prestazioni maggiori rispetto ai sistemi la cui funzione è centralizzata su unico elaboratore. Ciò è dovuto al fatto che il carico computazionale viene diviso tra i diversi nodi appartenenti al sistema. Questo tipo di architetture, inoltre, ci consentono di scalare orizzontalmente[3], in modo da poter mantenere costanti le prestazioni nonostante l'aumento delle richieste in arrivo.

1.3 Cloud Computing

Messo a fuoco il concetto di sistema distribuito, si passa ora ad analizzare quello di Cloud Computing. Il cloud computing, in ambito informatico, individua un meccanismo di erogazione di servizi da un fornitore ad un cliente attraverso la rete internet. Tali servizi possono essere di diversa natura, citiamo ad esempio l'archiviazione, la trasmissione o l'elaborazione di dati. Questi servizi sono messi a disposizione grazie all'assegnazione intelligente di risorse all'utente, attraverso procedure automatizzate.

Gli attori in gioco in questo meccanismo sono principalmente tre [1]:

- I Provider: si tratta dei fornitori dei servizi e delle infrastrutture. Essi mettono a disposizione, on demand, server, risorse di storage, database e software. Tra i principali provider ricordiamo Amazon Web Services, Google Cloud Platform e Azure;

[3] Lo scaling è la capacità di aumentare o diminuire le risorse in base alla necessità. Esiste lo scaling verticale che consiste nel ridimensionare le risorse di una singola macchina. Esiste poi lo scaling orizzontale che consiste nel ridimensionare il numero delle macchine coinvolte accendendo o spegnendo istanze in maniera opportuna.

- I clienti amministratori: coloro che scelgono e configurano i servizi offerti dai provider al fine di offrire servizi aggiuntivi. Esistono clienti amministratori di ogni tipologia, dimensione e settore. Si parte infatti dai produttori di videogames che utilizzano il cloud computing per realizzare videogiochi per utenti in tutto il mondo, passando dalle banche e dalle aziende che lo utilizzano per prevenire e mitigare attività fraudolente;
- I clienti finali: coloro che utilizzano i servizi configurati dai clienti amministratori.

1.3.1 IaaS, PaaS e SaaS

Esistono tre tipi di cloud computing [2]: Infrastructure as a Service, Platform as a Service, e Software as a Service. La discriminante si basa sul controllo che l'utente ha sulle risorse messe a disposizione

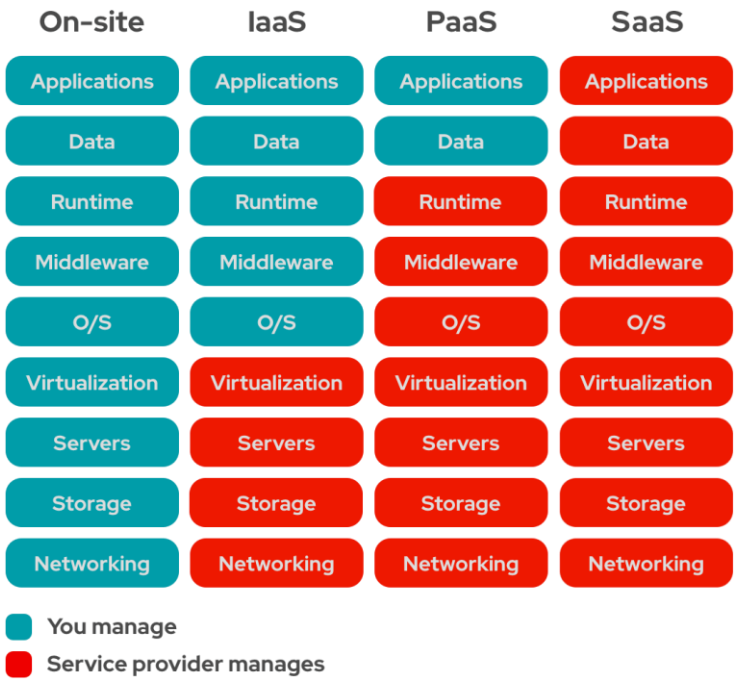


Figura 1.1: Tipologie di cloud computing e differenziazioni in termini di controllo delle risorse[4]

Infrastructure as a Service, IaaS, è il livello che fornisce maggiore flessibilità. In questo caso il provider mette a disposizione i servizi di infrastruttura prendendosi la responsabilità di gestire rete e virtualizzazione. L'utente ha il carico di installare i Sistemi

[4] Fonte: https://www.redhat.com/it/topics/cloud-computing/iaas-vs-paas-vs-saas

Operativi, i middleware[5] ed ha il quasi completo controllo sull'infrastruttura stessa, potendo interagirvi grazie all'utilizzo di apposite API o attraverso una dashboard. In questo caso, molto spesso il modello di consumo è *pay-as-you-go*.

Platform as a Service, PaaS, offre un livello di astrazione maggiore rispetto a Iaas, andando a togliere all'utente la responsabilità di gestione dell'infrastruttura sottostante. In questo modo, l'utente può concentrarsi soprattutto sulla distribuzione e sulla gestione delle applicazioni. La gestione dell'infrastruttura sottostante diventa, a questo punto, responsabilità del provider.

Software as a Service, SaaS, rende possibile all'utente, l'utilizzo di un software specifico senza che esso si preoccupi della gestione dell'infrastruttura su cui il esso poggia, responsabilità, chiaramente, delegata al Provider.

1.3.2 I vantaggi del Cloud

Tra i principali vantaggi del Cloud Computing ricordiamo:

- Risparmio economico: affidarsi ad un provider ed utilizzare la sua infrastruttura consente di avere dei costi decisamente minori rispetto a quelli che ne risulterebbero dalla messa in piedi di un datacenter autonomo;
- Flessibilità: il cloud permette di utilizzare, testare e verificare nuove tecnologie in maniera estremamente semplice e rapida. Questo consente all'utente di rimanere sempre al passo con i tempi;
- Scalabilità: grazie al cloud non è necessario allocare risorse sovrastimando i carichi di lavoro, andando dunque a pagare macchine che saranno per lo più inutilizzate. Potranno essere, piuttosto, utilizzate in ogni momento solo le macchine necessarie. Ciò consente di gestire i picchi aumentando le istanze delle macchine e i momenti di basso carico diminuendole. Questo si traduce in un chiaro risparmio economico;

[5] Il middleware è un componente molto comune nei sistemi distribuiti. Esso a livello di architettura si interpone tra il sistema operativo e l'applicazione, ed ha il compito di astrarre operazioni complesse e di fungere da intermediari tra diverse applicazioni.

- Distribuzione globale immediata: il cloud consente di distribuire i propri servizi su scala globale in pochissimo tempo.

1.4 Algo e requisiti progetto

Esaminati ed assimilati i concetti cardine sui quali poggia il lavoro svolto nel progetto, è ora giunto il momento di introdurre quelli che sono i requisiti dello stesso.

Algo è un'azienda torinese nata dall'intuizione di due ragazzi, Luca Gonnelli e Ilenia Notarangelo, i quali resisi conto della tendenza riguardante i contenuti multimediali, analizzata all'inizio del capitolo, si sono impegnati nella realizzazione di una piattaforma cloud, capace di creare video basati su dati in tempo reale, in maniera pressocché automatica. Ad oggi Algo è un'importante realtà che vanta numerose collaborazioni con importantissimi brand internazionali.

Dal punto di vista tecnico, la fase cruciale del loro lavoro è il rendering[6] dei video. Come afferma lo stesso Nina Farzaneh, Data Scientist all'interno di Algo, tale processo occupa circa il 90% del tempo di elaborazione completo di un video. Questa procedura avviene attraverso l'utilizzo di un programma di Adobe creato, appunto, per la realizzazione di animazioni ed effetti video. Si tratta di After Effects, e, nello specifico, per la fase di rendering, viene utilizzato una particolare modalità denominata Watch Mode, che si avvale di protocolli proprietari di Adobe, per parallelizzare il lavoro su più macchine; tuttavia, questa soluzione risulta essere obsoleta ed inadeguata, in quanto, soggetta a numerosi bug che ne rendono l'utilizzo complicato.

Ad evidenziare ancora di più quella che è l'inefficienza di questa soluzione, è l'analisi condotta da Algo riguardante l'utilizzo delle macchine che hanno a disposizione. Da tale analisi è infatti emerso che su cinque macchine, quattro rimangono completamente inutilizzate per il 99% del tempo. È inoltre necessario considerare che Algo utilizza come cloud provider Azure, dunque nella costruzione della soluzione, si è voluto dare continuità a questa scelta.

[6] Il rendering è una procedura attraverso la quale, partendo da un progetto, mediante software specializzati, vengono elaborati e poi concatenati i singoli fotogrammi al fine di creare un video completo.

Tra i requisiti del progetto, dunque, vi è sicuramente quello di andare a migliorare l'attuale infrastruttura di rendering, rendendola più efficiente sia dal punto di vista economico e sia dal punto di vista tecnico, aumentando anche la robustezza della stessa.

1.4.1 Carichi di lavoro

Al fine di comprendere meglio quale sia il carico di lavoro e quindi le effettive necessità dell'infrastruttura richiesta, sono state svolte delle analisi sui dati delle elaborazioni, dal 2019 fino ad oggi, forniti da Algo.

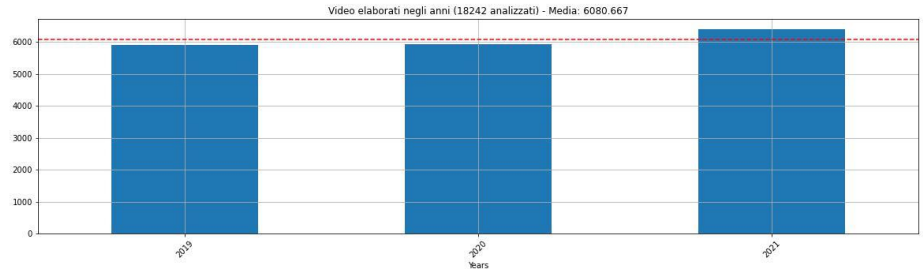


Figura 1.2: Analisi di video elaborati negli anni, svolta da David Lomuscio.

Dal grafico in Figura 1.2 si può vedere come il trend di video elaborati nel corso degli anni sia abbastanza stabile sulla cifra di circa 6000 video l'anno. Il periodo dell'anno in cui i sistemi sono maggiormente sotto pressione è costituito dai mesi di agosto, settembre e ottobre, mentre le fasce orarie di stress per il sistema sono per lo più quelle pomeridiane e la mezzanotte.

Dal punto di vista operativo però, l'analisi di maggior rilievo è quella relativa alle sovrapposizioni. Per sovrapposizione si intende l'elaborazione contemporanea di più video. Essa è stata calcolata considerando solo i video iniziati prima della fine dell'elaborazione di altri; quindi, quelli per cui i periodi di elaborazione si sovrappongono.

Come si evince dal grafico in Figura 1.3 e da altri grafici risultanti da altre analisi, il fenomeno della sovrapposizione è abbastanza raro. Si tratta di pochi casi nel corso degli anni, e la media, tenendo conto solo di questi casi, si attesta circa su 2,14. È stata inoltre estratta la distribuzione di queste sovrapposizioni nel corso delle ore del giorno e questa, in accordo con quanto emerso dall'analisi delle elaborazioni per lo stesso criterio, risulta

essere pressocché stabile, ad eccezione di un picco che si verifica nella fascia oraria delle 17.

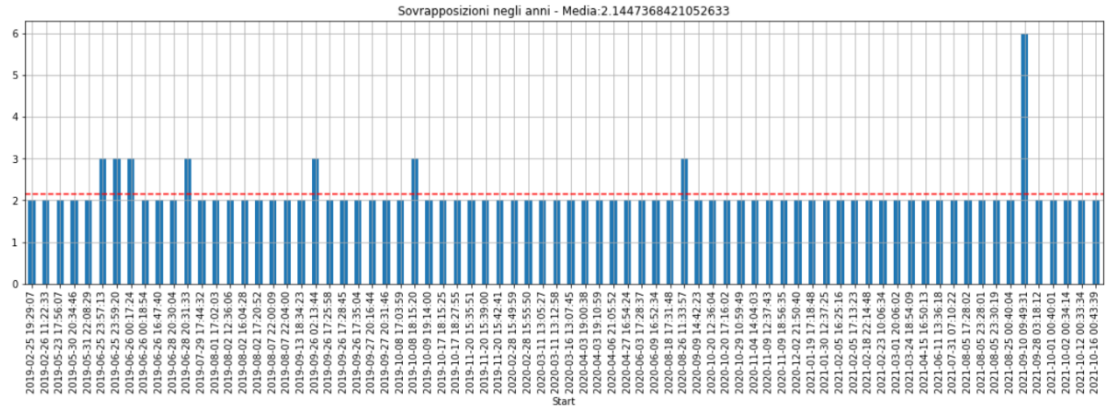


Figura 1.3: Analisi sulle sovrapposizioni di video nel tempo, svolta da David Lomuscio.

Anche le altre analisi sulle sovrapposizioni sono in linea con quelle fatte rispetto al criterio dei video elaborati. Si ha, ad esempio, una tendenza abbastanza stabile di distribuzione delle sovrapposizioni nel corso dei mesi, con un picco nei mesi di agosto, settembre e ottobre.

Infine, si è provato a trovare una sorta di fattore di amplificazione, un coefficiente che potesse permettere di prevedere quale fosse il tempo di elaborazione stimato, a partire dalla durata del video stesso. Tale analisi non ha portato ad un risultato preciso in quanto la durata dell'elaborazione dipende per lo più dalla complessità del video piuttosto che dalla sua durata. Come si evince dal grafico in Figura 1.4 il fattore di amplificazione risulta essere imprevedibile per video di breve durata, mentre pare attestarsi intorno al 5, per video di durata maggiore di 65 secondi.

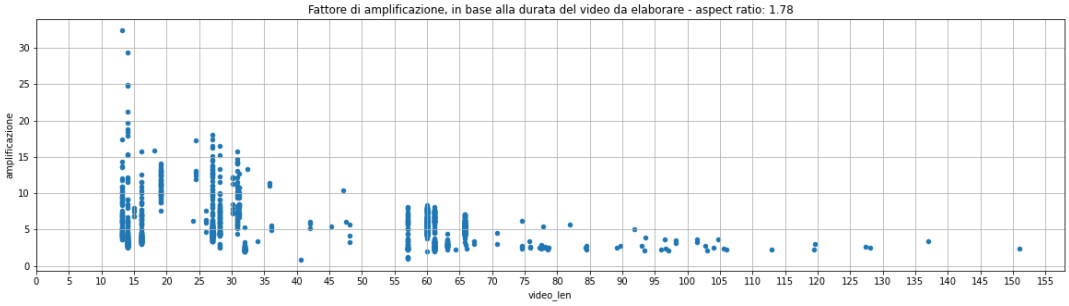


Figura 1.4: Analisi sul tempo richiesto per elaborare un video, svolta da David Lomuscio

Questo comportamento potrebbe essere dovuto al fatto che la percentuale di frame complessi tende a diminuire con l'aumentare della durata del video.

Dalle precedenti analisi emergono dunque le concrete necessità e gli effettivi requisiti dell'infrastruttura che deve essere costruita. Deve trattarsi di un'infrastruttura capace di fornire maggiore affidabilità, performance, e efficienza dal punto di vista economico. Che sia dunque capace di sfruttare appieno le risorse a disposizione, evitando di dover pagare risorse che possono rimanere per lo più inutilizzate.

Per quanto riguarda la scalabilità, invece, questa può essere implementata seguendo due paradigmi diametralmente diversi tra loro. Il primo paradigma è quello definito internamente come *scalabilità reattiva*. Attraverso questo paradigma si cerca di dimensionare le risorse, in termini di istanze di macchine virtuali, reagendo, appunto, al carico computazionale richiesto. Il secondo paradigma invece è quello che è stato definito come *scalabilità predittiva*, che consente di dimensionare le risorse in base a previsioni fatte a seguito dell'analisi della storia delle elaborazioni passate.

Considerando che i trend sembrano essere abbastanza chiari, e le analisi evidenziano modelli ricorrenti nel tempo, è stata scelta la scalabilità predittiva a discapito della scalabilità reattiva, la quale presenta anche dei limiti tecnici che verranno poi chiariti ed analizzati nei capitoli successivi.

Capitolo Secondo

Capitolo 2

2 Tecnologie

All'interno di questo capitolo verranno analizzate le numerose tecnologie studiate durante la fase di design e di sviluppo della soluzione. Alcune di esse non sono state poi utilizzate effettivamente all'interno del progetto, ma sono comunque meritevoli di menzione in quanto ne verranno poi studiati i limiti che hanno portato alle scelte fatte.

2.1 Cloud Native Computing Foundation

Nel capitolo precedente abbiamo compreso quanto sia importante e quali siano i vantaggi del Cloud. La Cloud Native Computing Foundation, CNCF, è un'organizzazione che sostiene questo approccio, mantenendo in piedi l'ecosistema che vi è alla base e salvaguardando l'accesso democratico alle tecnologie che lo costituiscono.

Una delle cause più rilevanti nella nascita di questa fondazione, è da ritrovarsi nella grande crisi mondiale del 2008 [3]. A seguito di tale crisi, infatti, le aziende hanno cercato di muoversi verso un assetto sempre più sostenibile ed efficiente, cercando di abbassare i costi e mantenendo comunque elevato il livello tecnologico. Tutto ciò ha portato ad un grande sviluppo delle soluzioni open source[7], le quali garantiscono, grazie al costante lavoro delle community, un alto livello tecnologico ad un prezzo decisamente più basso rispetto alle soluzioni proprietarie. Ad oggi, la maggior parte delle aziende, che lavorano nell'ambito tecnologico, adottano soluzioni open source.

Altro vantaggio da non sottovalutare, per quanto riguarda il mondo open source, è quello relativo allo sviluppo e alla crescita della soluzione. Questo aspetto è dovuto al fatto che alla base di tali soluzioni vi è la continua collaborazione tra utilizzatori e sviluppatori.

[7] Software libero da vincoli di copyright.

Esempio di quanto detto fino ad ora è Google, che nel 2014 rilascia Kubernetes, una delle maggiori soluzioni per quanto riguarda l'orchestrazione di container, e, resosi subito conto del fatto che, una community open source, avrebbe contribuito in maniera netta allo sviluppo della soluzione appena lanciata, decide, nel 2015, di stringere una partnership con Linux Foundation[8] dando vita, appunto, alla Cloud Native Computing Foundation. La CNCF in definitiva può essere descritta come quell'organizzazione che sostiene e mantiene progetti open-source e cloud native in tutto il mondo.

Figura 2.1: Logo Cloud Native Computing Foundation

La necessità nasce dal fatto che, al giorno d'oggi, anche le aziende che non sono nell'ambito software, devono occuparsi di questo aspetto. Dunque, CNCF permette, proprio a queste aziende di potersi avvicinare al mondo cloud in una maniera più rapida ed economica, fornendo anche delle certificazioni che permettono alle soluzioni di essere portabili, senza essere troppo legate al provider che mette a disposizione i servizi.

2.2 Kubernetes

Come accennato nel paragrafo precedente, Kubernetes, anche noto come *k8s*, è una piattaforma open source, che nasce nel 2014 dall'idea di Google di voler mettere insieme la sua esperienza pluriennale per quanto riguarda la gestione di carichi di lavoro su larga scala. In sintesi, Kubernetes è un sistema che consente di gestire cluster di nodi sui quali vengono eseguiti carichi di lavoro attraverso container [4].

Figura 2.2: Logo Kubernetes

[8] La Linux Foundation è l'organizzazione che mantiene lo sviluppo del kernel Linux

Per comprendere pienamente un sistema come Kubernetes, bisogna conoscere la storia evolutiva per quanto concerne l'ambito della distribuzione dei servizi. Tale storia può essere scomposta in tre fasi. Esse, in ordine cronologico, sono [4]:

- L'era dei server fisici, che prevedeva l'utilizzo di server fisici sui quali le applicazioni venivano eseguite, senza nessun modo per definire quali fossero i limiti delle risorse utilizzabili da ogni applicazione. Chiaramente questo approccio presenta dei problemi che vengono evidenziati soprattutto nel momento in cui si decide di eseguire contemporaneamente più applicazioni sullo stesso server fisico. In questo caso, infatti, potrebbero esserci applicazioni che utilizzano più risorse rispetto a quelle necessarie, andando a toglierle dalla disponibilità di altri processi. Una soluzione a questo tipo di problematica è quella di predisporre un server fisico per ogni applicazione, ma questa risulta essere poco flessibile e assai costosa.
- L'era delle VM prevede l'utilizzo delle Virtual Machine[9] per superare i limiti osservati nella fase precedente. Attraverso le VM, dunque si riesce ad isolare completamente un'applicazione da un'altra andando ad assegnare correttamente le risorse e rendendo l'ambiente più sicuro, anche considerando il fatto che l'isolamento non permette l'interferenza tra le diverse applicazioni. Questo approccio, per quanto potente e flessibile, sebbene sia ancora utilizzato, presenta il limite dovuto all'overhead di ogni sistema operativo installato sulle Virtual Machine.
- L'era dei Container, invece, prevede l'utilizzo di Container, appunto, i quali sono, dal punto di vista operativo, simili alle macchine virtuali, ma forniscono un isolamento meno stringente. Questo è dovuto al fatto che condividono con l'host[10] il sistema operativo. Per questo motivo i container superano il limite caratteristico della fase precedente, ovvero quello relativo all'overhead del sistema operativo.

[9] Una macchina virtuale è definita come un software che attraverso tecniche di virtualizzazione riesce a costruire un ambiente che emula le caratteristiche di una macchina fisica.
[10] L'host è la macchina fisica che ospita ed esegue le VM e i Container

2.2.1 Container

I container sono delle unità software attraverso le quali è possibile impacchettare del codice applicativo assieme a tutte le sue dipendenze, ed eseguirlo in maniera rapida [5]. Come visto precedentemente, per mezzo dei container è possibile ottenere isolamento tra le varie applicazioni, senza introdurre overhead dovuto all'installazione di sistemi operativi addizionali.

Dal punto di vista tecnico, è molto importante, nell'ambito dei container, il concetto di *namespace*. Si può infatti affermare che il sistema operativo host isola i container proprio grazie ai namespace. Essi possono essere visti come tabelle del kernel che rappresentano le unità logiche che il sistema operativo si trova a dover gestire. La creazione di diversi namespace è stata resa possibile, nei sistemi operativi Linux, attraverso l'introduzione della system call[11] *clone()*, la quale, a differenza della più classica *fork()*[12], crea un nuovo processo all'interno di un nuovo namespace. Ogni qualvolta che il sistema operativo viene chiamato in causa, ad esempio attraverso una system call, la richiesta viene interpretata secondo quello che è il namespace di chi l'ha effettuata. Ad esempio, per quanto riguarda la system call relativa all'apertura di un file, fornito un *path*, questo viene interpretato in base al namespace di chi ha richiesto l'esecuzione di tale operazione. È possibile, dunque che a *path* uguali, corrispondano file diversi se la richiesta dovesse provenire da namespace differenti.

Altro aspetto tecnico fondamentale si osserva nella maniera in cui i container gestiscono il *file system*. A differenza dell'approccio utilizzato dalle Virtual Machine, in cui ogni macchina ha la necessità di avere un proprio disco virtuale, i container, al fine di ridurre lo spazio di archiviazione richiesto, utilizzano il cosiddetto *layered file system*. Il *layerd file system* è un *file system* costituito da più strati, ogni strato contiene un set di cartelle e file che estendono, e in alcuni casi sovrascrivono il set contenuto nello strato inferiore. Sarà poi compito del driver del filesystem fornire una versione unificata dello stesso. Un

[11] La system call è il meccanismo attraverso il quale un processo a livello utente richiede l'esecuzione di un'operazione a livello kernel.
[12] La fork() è una system call di Linux che permette la creazione di un nuovo processo.

container è costituito da diversi layer *read-only*, e un solo layer *read-write*, quello più in cima.

2.2.1.1 Docker

Docker è una tecnologia open source disponibile per Linux, Windows e MacOS, che sfrutta i container. Essa si basa sull'esistenza di quattro componenti principali:

Figura 2.3: Logo Docker

- Docker Engine, processo che si occupa dell'esecuzione dei container. Può essere identificato come un demone che permette all'utente di interagire con i servizi Docker attraverso un'API. Esso inoltre standardizza la gestione dei container garantendo portabilità.
- Docker Client, componente che fornisce accesso alle funzionalità offerte dall'API del Docker Engine. È disponibile sia da linea di comando, sia come interfaccia e sia come componente web[13].
- Docker Image, può essere definito come un *template read-only* contente i *file system layer* necessari per far partire un container. Una Docker Image può anche basarsi su altre Docker Image. Si parla anche di *Dockerfile*, un file contenente tutte le informazioni necessarie per assemblare una Docker Image.
- Docker Container, è identificabile come un'istanza eseguibile di una Docker Image. Quando viene lanciato un container a partire da un'immagine, viene aggiunto, ai *file system layer, read-only* dell'immagine, un nuovo strato *read-write*.

Analizzati i componenti fondamentali di Docker, resta da comprendere come siano organizzate le immagini all'interno di questo sistema. Esse si strutturano in *repository*, che a loro volta, "vivono" all'interno di *registry*. Ogni host Docker possiede una propria *registry* locale. Risulta possibile accedere a *registry* remote come, ad esempio, quella ufficiale di Docker, *Docker Hub*.

[13] Attraverso l'utilizzo di Portainer.

2.2.1.2 Volumi e Networking

I container possono montare, nel file system dell'immagine, cartelle presenti nel file system dell'host. Questo, è reso possibile grazie all'utilizzo dei Volumi. Per mezzo di essi, inoltre, è ragionevole pensare di condividere porzioni di file system, non solo tra host e container, ma soprattutto tra container diversi.

Per quanto concerne il networking invece, la situazione è un po' più complessa. Sebbene infatti, ogni container possa essere facilmente accessibile dall'intero stack di rete, l'isolamento, dovuto ai namespace, rende i container inaccessibili dall'esterno; tuttavia, sono comunque stati introdotti dei meccanismi che permettono di configurare in maniera efficiente e flessibile il sottosistema di rete.

Ogni container, di default, può aprire una connessione in uscita. Ogni qualvolta che si avvia un container, ad esso viene assegnato un indirizzo IP privato[14] che viene istradato verso l'host attraverso le *iptables*. La maniera più semplice, per abilitare un container a fornire un servizio di rete, consiste nel mappare una porta dell'host con una corrispondente porta del container. Per interconnettere fra loro due container differenti, invece, esistono due approcci distinti:

- Il primo consiste nel configurare una connessione diretta fra le parti;
- Il secondo consiste nel configurare i container come appartenenti ad una virtual subnet, specificando le funzioni che devono essere applicate tra le varie subnetwork.

Nel caso specifico di Docker, nella fase di installazione, vengono create automaticamente tre reti:

- *Bridge*, per cui, tutti i container che vi appartengono, sono abilitati a comunicare attraverso l'IP ricevuto all'avvio;
- *None*, per cui, i container connessi a tale rete sono abilitati ad usare solo l'interfaccia di loopback;

[14] Un IP privato è un indirizzo IP appartenente a determinate classi, le quali sono riservate alle reti locali, al fine di ridurre la richiesta di indirizzi IP pubblici.

- *Host*, per cui, tutti i container connessi ad essa condividono l'intero stack di rete con l'host.

Docker consente anche di creare configurazioni di rete personalizzate di due tipi: *Bridge Network* e *Overlay Network*.

2.2.2 Componenti Kubernetes

Per analizzare i componenti che sono alla base del funzionamento di Kubernetes, è necessario partire dal concetto di *cluster*. Un *cluster* è un insieme di nodi che sono coordinati tra loro al fine di distribuire un'elaborazione più complessa. In un cluster Kubernetes esiste una distinzione tra: nodi appartenenti al *control plane* e *worker nodes*. Il rapporto che vi è tra le due tipologie di nodo è il seguente: i nodi appartenenti al *control plane* gestiscono i worker nodes, prendendo decisioni e reagendo a determinati eventi [6]. I *worker nodes,* invece, sono responsabili di eseguire i carichi di lavoro effettivi. Per interagire con il cluster Kubernetes da linea di comando è possibile utilizzare il tool "kubectl".

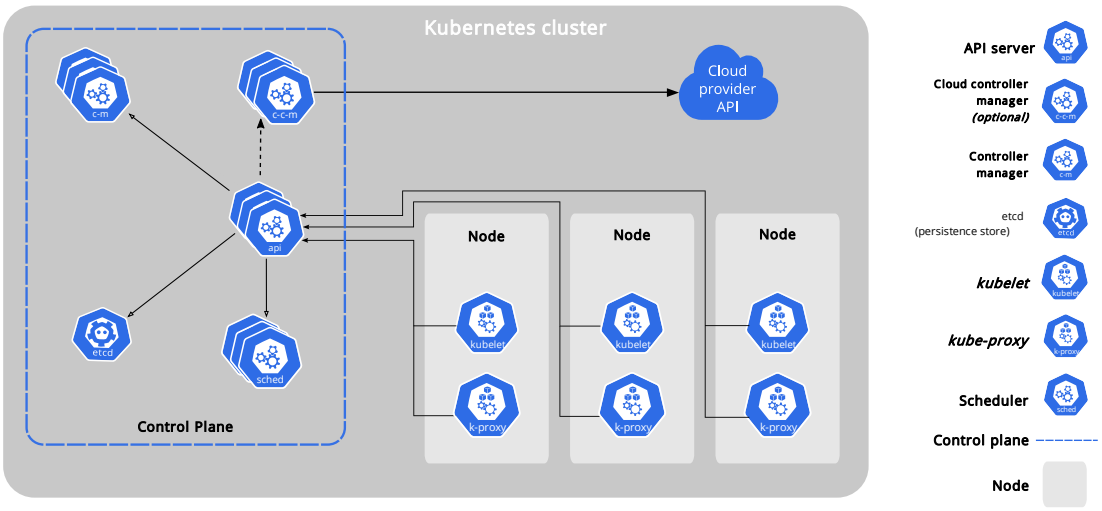


Figura 2.4: Schema dei componenti di un cluster Kubernetes[15]

Come si evince dalla Figura 2.4, i componenti appartenenti al *control plane*, e dunque eseguiti solo sui nodi riguardanti il piano di controllo, sono [6]:

[15] Fonte: https://kubernetes.io/docs/concepts/overview/components/

- *kube-apiserver*, componente responsabile di gestire l'interazione con il control plane del cluster Kubernetes, attraverso l'esposizione di endpoint;
- *kube-scheduler*, componente incaricato di assegnare i carichi di lavoro ai diversi nodi presenti nel cluster;
- *etcd*, identificabile come una sorta di database di tipo *key-value*, utilizzato come supporto di *storage*;
- *kube-controller-manager*, componente che si occupa di gestire i vari controllori[16];
- *cloud-controller-manager*, componente simile al *kube-controller-manager*, ma che si occupa dell'interazione e della gestione di componenti per cui è responsabile il *cloud provider*.

Sempre in riferimento alla Figura 2.4, si possono ora analizzare i *node components*, ovvero quei componenti che vengono eseguiti su tutti i nodi presenti nel cluster. Essi sono [6]:

- *kubelet*, identificabile come quell'entità che si preoccupa della corretta esecuzione dei carichi di lavoro, valutandone l'integrità e la reattività;
- *kube-proxy*, componente responsabile della gestione dei servizi Kubernetes, i quali saranno dettagliati meglio nei paragrafi successivi;
- *Container runtime*, il software responsabile dell'esecuzione dei container all'interno delle macchine. Kubernetes supporta diversi *container runtime*, tra i quali ricordiamo ad esempio *containerd* e *Docker*.

2.2.3 Risorse Kubernetes

Le risorse sono gli elementi logici fondamentali di Kubernetes. Grazie ad esse è possibile definire quali siano e come debbano essere organizzati i diversi carichi di lavoro. Per definire una risorsa è necessario costruire un file *manifest* ed applicarlo al cluster.

Tutte le risorse hanno la medesima struttura:

- Tipo dell'oggetto, che è definibile univocamente tramite i campi *apiVersion* e *kind*;

[16] I controllori sono dei componenti atti a controllare lo stato del cluster. Nel caso in cui si si trovi in uno stato non contemplato, essi sono capaci, attraverso l'apiserver, di riportare la situazione desiderata.

- Dettagli di contorno dell'oggetto, identificabili tramite i campi *labels* e *annotations;*
- *Spec*, attraverso il quale si definisce lo stato desiderato per la risorsa;
- *Status*, attraverso il quale si può osservare lo stato corrente della risorsa.

2.2.3.1 Pod, Deployment, ReplicaSet e DeamonSet

Si passa dunque ad analizzare, quelli che sono, i più piccoli elementi costitutivi di una infrastruttura Kubernetes, i pod. Un pod è un set costituito da uno o più container che vengono eseguiti nello stesso contesto [7]; dunque, essi possono condividere risorse di storage e rete. Per comprendere meglio questa dinamica, in riferimento a quanto detto nei paragrafi precedenti, si può affermare che più container appartenenti allo stesso pod, vengono eseguiti sotto lo stesso namespace.

Dal punto di vista organizzativo, esistono due approcci per quanto riguarda la relazione che vi è tra pod e container. Un primo approccio è quello per cui si decide di assegnare ad ogni pod, un solo container. Tale metodologia risulta essere quella di più comune utilizzo, ed è in contrapposizione con il secondo approccio, che invece prevede l'esecuzione di più container all'interno del singolo pod. Nel secondo caso, i pod eseguono i container a seconda di come essi vengono dichiarati nel *manifest*[17] di definizione del pod stesso. È infatti possibile definire container classici piuttosto che *initContainer* e *ephemeralContainer*.

Gli *initContainer* sono i primi container ad essere eseguiti e la partenza dei container dichiarati come classici è subordinata alla loro corretta terminazione. Gli *ephemeralContainer* sono invece dei container utili soprattutto a fini di debug, in quanto essi non hanno nessuna garanzia riguardo alle risorse che possono utilizzare e non vengono praticamente mai riavviati. È possibile, inoltre, eseguire container in modalità privilegiata, il che consente al container di avere accesso alle risorse dell'host, attraverso l'apposito flag *privileged* per quanto riguarda i container Linux, oppure attraverso l'utilizzo dei Windows HostProcess per quanto concerne i container basati sul sistema operativo di Microsoft.

[17] File di tipo YAML o JSON, utile a dichiarare quale sia lo stato desiderato all'interno del cluster Kubernetes.

Esistono delle risorse che consentono di gestire l'esecuzione di più pod attraverso l'utilizzo dei *controller*. Tali risorse sono: i *Deployment*, che consentono di gestire le repliche di uno specifico pod all'interno del cluster; gli *StatefulSet*, che consentono di gestire le repliche e lo scaling dei pod che hanno dello storage persistente; i *DeamonSet*, che assicurano la presenza dei pod specificati in ogni macchina del cluster.

Una delle caratteristiche dei pod è il fatto che essi sono concepiti per essere effimeri. Una volta schedulati, rimangono a carico del nodo fin quando:

- Terminano la loro esecuzione;
- Vengono rimossi a causa della grande quantità di risorse che stanno utilizzando;
- La macchina su cui si trovano, per un qualsiasi motivo, si spegne.

La trattazione prosegue ora con l'analisi del ciclo di vita dei pod. Durante la loro esecuzione, i pod attraversano diverse fasi:

1) Nel momento in cui viene creato, il pod entra in una fase detta *pending*, nella quale rimane in attesa si essere schedulato[18];
2) Appena schedulato esso fa partire i container al suo interno ed entra nella fase di *running*. In questa fase vi è, all'interno del pod, almeno un container in fase di esecuzione;
3) Quando tutti i container terminano, il pod può attraversare due fasi distinte, o la fase di *succeeded*, quando tutti i container al proprio interno terminano con successo, oppure la fase di *failed*, quando i container al proprio interno terminano con errore.
4) Esiste una quarta fase, detta di *Unknow*, riscontrabile quando, per qualsiasi motivo non si riesce a risalire all'attuale stato del pod.

È inoltre possibile specificare, per ogni pod, quali siano le risorse minime e massime richieste, in termini di CPU e di memoria, attraverso degli appositi campi indicabili nel *manifest* della risorsa.

[18] Un pod si definisce schedulato quando lo scheduler lo ha assegnato ad un nodo specifico.

2.2.3.2 Service

Altra risorsa fondamentale all'interno dell'ecosistema Kubernetes, sono i *Service*. I *Service* risultano essere molto importanti perché ci consentono di esporre un gruppo di pod sulla rete. Una delle caratteristiche dei pod, analizzate nel paragrafo precedente, è rappresentata dal fatto che essi sono progettati per essere effimeri.

Come si è visto, è comunque possibile controllare l'esecuzione di pod proprio grazie ai controller, i quali si assicurano che quelli in esecuzione rispettino i vincoli dettati dal tipo di controller stesso. Se ad esempio, un deployment specifica l'attività di due pod specifici, il *kube-controller-manager* si assicura che in ogni momento siano in esecuzione esattamente due pod indicati. Se uno dei due pod dovesse terminare con errore, sarebbe compito del controller ristabilire lo stato desiderato andando ad avviare un altro pod con le stesse caratteristiche, in modo da sostituire quello appena concluso. Il controller, dunque, non va ad avviare lo stesso pod, ma applica una semplice sostituzione. Nel momento in cui un pod viene avviato, viene assegnato ad esso un indirizzo IP; dunque, non risulta essere necessariamente vero che, considerando l'esempio sopracitato, il nuovo pod abbia lo stesso indirizzo IP di quello appena concluso.

In alcuni casi, quelli in cui è imprescindibile la collaborazione tra i pod tramite la rete, questo scenario non è ottimale, in quanto sarebbero necessari dei meccanismi aggiuntivi per riuscire a reperire, in qualsiasi momento, l'informazione relativa all'indirizzo IP del pod con cui si vuole comunicare.

È proprio per questa ragione che sono stati introdotti, all'interno di Kubernetes, i *Service*. Grazie ad essi, è possibile raccogliere gruppi di pod, attraverso la meccanica dei selettori[19], ed assegnare a tali gruppi indirizzi IP statici. È inoltre possibile, attraverso i *service*, eseguire una sorta di load balancing affinché richieste in arrivo per una determinata applicazione, siano smistati in maniera equa, tra tutti i pod che eseguono istanze di tale applicazione, così da non sovraccaricare eccessivamente determinati pod.

È possibile creare quattro tipi di *service*, essi sono:

[19] I selettori sono dei meccanismi attraverso i quali si riescono a identificare risorse o gruppi di risorse per mezzo di etichette.

- *ClusterIP*, in cui la porta specificata, risulta essere accessibile solo all'interno del cluster;
- *NodePort,* consente di esporre una risorsa sull'indirizzo IP del nodo su cui è ospitata, alla porta indicata;
- *LoadBalancer*, che rende la risorsa accessibile anche dall'esterno del cluster ed implementa un load balancer;
- *ExternalName*, utilizzato per mappare un servizio su un indirizzo esterno. Questa tipologia risulta essere poco utilizzata.

Altra dinamica importante, se messa in relazione al concetto di *service* è la *readinessProbe*, la quale consiste in continui test che vengono effettuati sul pod, attraverso endpoint definiti, al fine di capire se esso è effettivamente pronto ad erogare un servizio. Nel caso di fallimento della *readinessProbe* il pod verrebbe escluso dai candidati a ricevere le richieste smistate dal load balancer.

2.2.3.3 La gestione dei Volumi

Un'altra risorsa utile da analizzare, al fine di comprendere al meglio l'architettura presentata nei capitoli successivi, è rappresentata dai volumi. L'esigenza di inserire i volumi in Kubernetes nasce dal fatto che i container, in quanto effimeri, non hanno un meccanismo di persistenza del dato. Quando un container termina l'esecuzione, il contenuto del suo disco di archiviazione viene eliminato. Questa situazione potrebbe non essere ottimale, soprattutto per la costruzione di applicazioni più complesse.

Dal punto di vista concettuale, e analogamente a quanto detto nei paragrafi precedenti per i volumi Docker, un volume Kubernetes è una cartella presente su un nodo del cluster, accessibile dai container all'interno dei pod. Tali volumi possono essere montati da più container contemporaneamente, e a sua volta un container può montare più volumi, non risulta tuttavia possibile, montare dei volumi in un ramo di un altro volume.

Ai fini del progetto sono stati esplorati nello specifico i P*ersistent Volume PV* e i *Persistent Volume Claim PVC*. I primi sono identificabili come degli spazi di archiviazione messi a disposizione dall'amministratore del cluster o ottenuti per mezzo

delle *Storage Class*[20]. Essi sono completamente indipendenti dal ciclo di vita dei pod. I *PVC*, invece, rappresentano una richiesta di spazio di archiviazione da parte dell'utente

2.2.3.4 Altre risorse

Sebbene non siano estremamente utili ai fini della comprensione dell'architettura proposta per il progetto, come si evince dalla figura 1.3, sono presenti, e meritano menzione, anche altre risorse utili al funzionamento di un'infrastruttura Kubernetes. Esse sono:

- Namespace: essi consentono di introdurre il concetto di "cluster logico". I namespace permettono di isolare completamente alcune risorse da altre fornendo una sorta di ordine logico;
- Horizontal Pod Autoscaler: una risorsa che consente di semplificare lo scaling orizzontale dei pod, ridimensionando in base a metriche classiche, quali ad esempio l'utilizzo medio di memoria e CPU, o anche in base a metriche custom definite dall'utente;
- ConfigMap e Secrets: due risorse molto utili per passare ai pod eventuali variabili d'ambiente;
- Ingress: risorsa che consente di rendere accessibile un pod attraverso un URL.

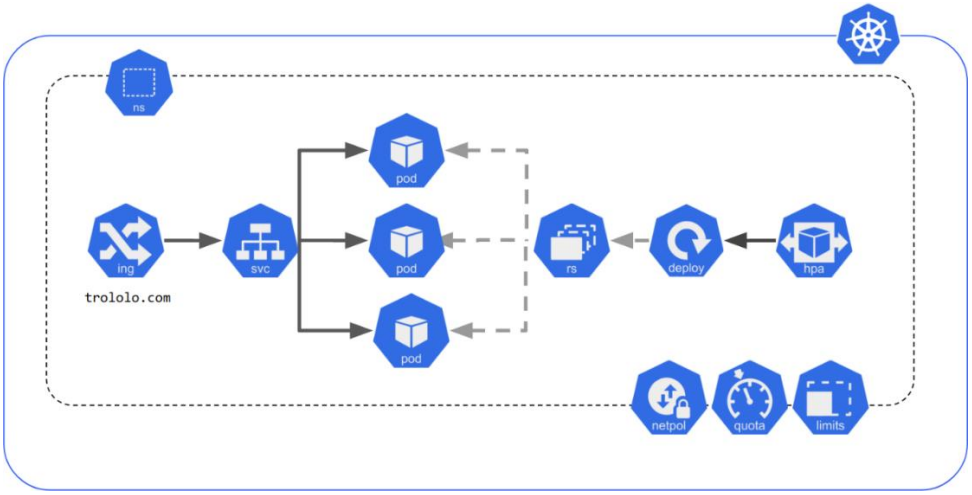


Figura 2.5: Rappresentazione dei principali componenti di Kubernetes[21]

[20] Le Storage Class sono dei meccanismi che permettono all'amministratore del cluster di differenziare lo storage in classi, ognuna con le proprie policy.
[21] Fonte: https://github.com/kubernetes/community/blob/master/icons/examples/schemas/std-app.png

È inoltre possibile estendere le funzionalità di Kubernetes attraverso la definizione di *Custom Resources* e attraverso l'utilizzo dell'*Operator Pattern*. Tale pattern consiste nello scrivere del codice, tipicamente in Golang[22], al fine di specificare quali debbano essere le operazioni da eseguire per reagire a determinati eventi.

2.3 Azure Kubernetes Service

Azure Kubernetes Service, noto anche come AKS, rappresenta l'implementazione Microsoft di Kubernetes, fornita da Azure come Platform as a Service. Essa risulta essere pressoché fedele alla versione vanilla[23] di Kubernetes, ma l'aspetto in cui risulta estremamente utile, riguarda la gestione delle VM generate nell'ambiente Azure, macchine virtuali sulle quali verranno poi eseguiti i pod.

È possibile, attraverso la dashboard di Azure, creare un cluster AKS e collegarvi il contesto relativo a kubectl in maniera semplice ed immediata attraverso il tool da linea di comando "az". Successivamente risulta essere possibile interagire con il cluster sia utilizzando il tool da linea di comando e sia utilizzando la dashboard fornita da Azure.

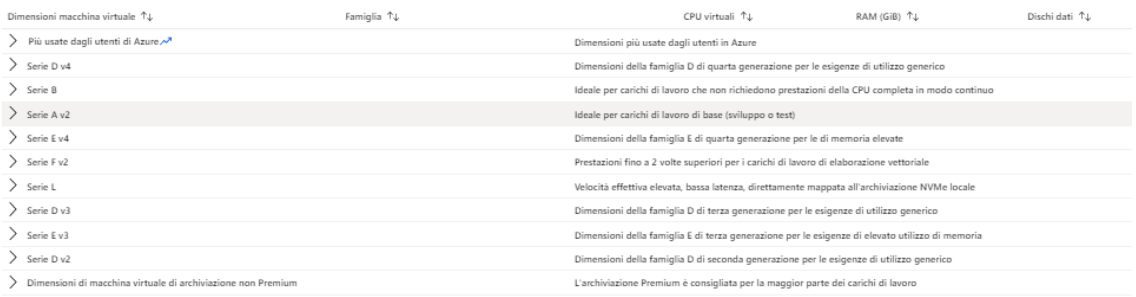
La configurazione del cluster è relativamente semplice e strutturata. Nella fase di creazione è possibile specificare quale debba essere il nome del cluster, il numero e la tipologia dei nodi appartenenti ad esso ed altre informazioni riguardanti la sicurezza, la rete, i metadati e le integrazioni con altri servizi quali ad esempio Azure Container Registry[24].

Per quanto riguarda le scelte che coinvolgono i nodi, Azure mette a disposizione dell'utente un'infinità di tipologie, tra cui è possibile effettuare la scelta. Da un punto di vista logico i nodi sono organizzati nei cosiddetti "pool di nodi". Caratteristica fondamentale dei pool di nodi è che tutti i nodi appartenenti allo stesso pool sono uguali per tipologia e per sistema operativo installato. È inoltre possibile decidere, per ogni pool, se esso debba essere costituito da un numero fisso di VM, oppure se possa scalare in base alle necessità in quanto a carico computazionale; in questo caso, risulta essere impostabile

[22] Golang è un linguaggio di programmazione sviluppato da Google.
[23] Un Software si dice vanilla quando esso non è stato modificato rispetto alla versione originale.
[24] Azure Container Registry è un servizio che consente di avere una registry Azure privata.

un range, attraverso il quale si può indicare il numero massimo ed il numero minimo di istanze utilizzabili. Un pool di nodi può poi essere identificato attraverso delle *label*, utili per sfruttare i selettori.

Come detto e come si evince dalla Figura 2.6, il bacino da cui scegliere la tipologia dei nodi utilizzabili per la costruzione di un pool è molto vasto. Esistono differenti tipologie ognuna delle quali ottimizzate per diversi carichi di lavoro.



Dimensioni macchina virtuale ↑↓	Famiglia ↑↓	CPU virtuali ↑↓	RAM (GiB) ↑↓	Dischi dati ↑↓
Più usate dagli utenti di Azure		Dimensioni più usate dagli utenti in Azure		
Serie D v4		Dimensioni della famiglia D di quarta generazione per le esigenze di utilizzo generico		
Serie B		Ideale per carichi di lavoro che non richiedono prestazioni della CPU completa in modo continuo		
Serie A v2		Ideale per carichi di lavoro di base (sviluppo o test)		
Serie E v4		Dimensioni della famiglia E di quarta generazione per le di memoria elevate		
Serie F v2		Prestazioni fino a 2 volte superiori per i carichi di lavoro di elaborazione vettoriale		
Serie L		Velocità effettiva elevata, bassa latenza, direttamente mappata all'archiviazione NVMe locale		
Serie D v3		Dimensioni della famiglia D di terza generazione per le esigenze di utilizzo generico		
Serie E v3		Dimensioni della famiglia E di terza generazione per le esigenze di elevato utilizzo di memoria		
Serie D v2		Dimensioni della famiglia D di seconda generazione per le esigenze di utilizzo generico		
Dimensioni di macchina virtuale di archiviazione non Premium		L'archiviazione Premium è consigliata per la maggior parte dei carichi di lavoro		

Figura 2.6: Screen relativo alla grande tipologia di VM utilizzabili per la creazione di nodi

Caratteristiche importanti, ai fini del progetto, di ogni tipologia sono:

- La famiglia, la quale specifica quale sia l'utilizzo per cui la VM è ottimizzata;
- CPU virtuali e RAM, che specificano quante siano le CPU e la dimensione della RAM della VM;
- Numero massimo di operazioni di I/O al secondo, caratteristica che risulta essere molto importante per quanto riguarda il pull delle diverse immagini contenute nei pod;
- Costo, che chiaramente risulta essere una caratteristica fondamentale al fine di scegliere in maniera opportuna.

Chiaramente è possibile creare pool di nodi Linux, i quali possono essere caratterizzati da un nome di al massimo 12 caratteri, piuttosto che un pool di nodi Windows, i quali possono essere caratterizzati da un nome di dimensione massima pari a 6 caratteri. Da alcuni test effettuati, come facilmente prevedibile, l'accensione di una macchina Windows richiede molto più tempo rispetto all'accensione di una corrispondente Linux.

Per concludere questa breve parentesi riguardante AKS, si può analizzare un caso verosimile di scaling dei nodi. Si immagini di trovarsi in una situazione tale da avere un pool di nodi, e un set di pod che consumano tutte le risorse a disposizione; nel momento

in cui si prova a creare un nuovo pod, su tale pool di nodi, lo scheduler, rileva l'impossibilità di procedere con l'operazione in quanto le risorse disponibili non bastano. A questo punto viene abilitato il meccanismo di scaling dei nodi, il quale può avere esito positivo, o esito negativo nel caso in cui il pool di nodi in analisi non sia abilitato allo scaling automatico oppure abbia raggiunto il numero massimo di nodi istanziabili. Viceversa, nel caso in cui ci si trovi ad avere una grande quantità di nodi, all'interno del pool, sotto una determinata soglia di utilizzo delle risorse, il sistema rileva questa condizione ed esegue uno scaling atto a ridurre il numero delle istanze attive.

2.4 KubeMQ

2.4.1 Panoramica comunicazioni tra microservizi

Aspetto fondamentale, per quanto riguarda la realizzazione di architetture di elaborazione distribuita, è caratterizzato dalle interazioni che possono esserci tra le varie componenti di questa infrastruttura. Tipicamente, in questi casi, si predilige avere un approccio che possa rendere i microservizi[25] in gioco *loosely coupled*, dunque, che possano svilupparsi in maniera indipendente l'uno dall'altro. Si prova a raggiungere questo obiettivo cercando di gestire la granularità degli stessi, nella fase di design, avendo ben chiaro che, una granularità troppo fine, può portare ad avere dei tempi di esecuzione maggiori, portando ad una crescita del numero delle interazioni richieste. La comunicazione tra i diversi servizi risulta quindi imprescindibile al fine di ottenere un sistema che possa evolversi in maniera più semplice, ma che allo stesso tempo riesca a svolgere correttamente il proprio lavoro.

Un altro aspetto fondamentale riguarda poi la dimensione dei messaggi che rendono possibile tale comunicazione. Alla luce del fatto che questi messaggi vengono scambiati attraverso la rete internet, è importante stabilire quale sia il giusto trade-off tra la dimensione dei messaggi e il numero dei messaggi da scambiare al fine di eseguire un'operazione. Risulta dunque palese che un messaggio di grandi dimensioni, riesce a

[25] I microservizi sono identificabili come dei servizi indipendenti e di piccole dimensioni che collaborano tra loro al fine di fornire un servizio completo.

trasportare un numero di informazioni maggiori rispetto ad un messaggio di piccole dimensioni; tuttavia, un messaggio di grandi dimensioni va sicuramente a consumare maggiore banda, trasportando informazioni, magari inutili, rispetto ad un messaggio di piccole dimensioni.

Esistono due approcci, che possono essere seguiti, nella fase di design e implementazione della comunicazione tra diversi microservizi. Essi sono:

- Approccio Sincrono: che prevede che un microservizio invii un messaggio ed attenda la risposta ad esso prima di continuare ad eseguire altre operazioni. Esempi di tale approccio sono le comunicazioni che si basano sul protocollo HTTP;
- Approccio Asincrono: che prevede che un microservizio invii un messaggio disinteressandosi completamente della risposta ad esso e, dunque, continuando a svolgere altre operazioni. La risposta verrà poi gestita attraverso delle callback. Esempi di tale approccio sono le comunicazioni Message Oriented.

Un approccio sincrono prevede che ci siano delle operazioni bloccanti. Tipicamente il client che contatta il server attraverso una comunicazione sincrona, blocca il thread nel quale viene eseguita questa operazione e lo sblocca solo nel momento in cui arriva una risposta da parte del server, oppure alla scadenza di un timeout. Chiaramente questo approccio presenta dei limiti evidenti, tra cui sicuramente il fatto di dover bloccare un thread per un tempo indefinito.

Tabella 1: Analisi dei tempi di attesa di un thread in diversi casi[26]

System event	Actual Latency	Scaled Latency
One CPU cycle	0.4 ns	1 s
Level 1 cache access	0.9 ns	2 s
Level 2 cache access	2.8 ns	7 s
Level 3 cache access	28 ns	1 min
Min memory access (DDR DIMM)	~100 ns	4 min
Intel Optane DC persistent memory access	~350 ns	15 min
Intel Optane DC SSD I/O	< 10 µs	7 hrs
NVMe SSD I/O	~25 µs	17 hrs
SSD I/O	50-150 µs	1.5-4 days
Rotational disk I/O	1-10 ms	1-9 months
Internet SF to NYC	65 ms	5 years

[26] Fonte: "Asynchronous programming" di Kirill Bobrov, september 2020

Come si evince dalla Tabella 1, se il colpo di clock di un processore fosse paragonato ad un secondo di vita di un essere umano, un'operazione che richiede l'accesso alla rete, costerebbe a tale processore circa 5 anni [8]. Tale esempio è stato fatto al fine di comprendere quanto sia importante non disperdere questi tempi di attesa, e dunque, muoversi verso un approccio asincrono.

2.4.2 Caratteristiche KubeMQ

KubeMQ è un progetto open-source facente parte del CNCF Landscape. La caratteristica principale di KubeMQ è quella di essere *message broker* e coda di messaggi nativa dell'ambiente Kubernetes. Esso supporta tutti i principali pattern di messaggistica esistenti, garantisce transazionalità e risulta, inoltre, essere facilmente installabile nei sistemi Kubernetes [9].

Figura 2.7: Logo KubeMQ

Il meccanismo di messagistica di KubeMQ si basa sull'individuazione di due entità fondamentali per la comunicazione: il mittente, colui che invia i messaggi verso una o più destinazioni, e il destinatario, colui che riceve i messaggi e può iscriversi a determinati *channel*, al fine di ricevere determinate informazioni.

2.4.2.1 Channel

Un *channel* è una stringa che individua logicamente un gruppo di destinatari interessati al messaggio su cui questo viene inviato. Esistono diverse caratteristiche tipiche di questa stringa. Essa non deve essere vuota, è *case insensitive* e non sono ammessi spazi. I diversi *channel* possono seguire un'organizzazione gerarchica, i livelli di tale gerarchia sono identificabili attraverso il carattere ".". Esistono poi due caratteri particolari che identificano gruppi di canali appartenenti allo stesso livello gerarchico o a livelli gerarchici differenti. Tali caratteri sono detti *wildcard*. Essi sono:

- "*": identifica tutti i canali esattamente appartenenti al livello gerarchico che caratterizza la posizione dell'asterisco. Ad esempio "A.*" va a coinvolgere i canali "A.B" e "A.C" ma non il canale "A.B.C" o "B.C";

- ">": coinvolge tutti i canali appartenenti sia al livello gerarchico individuato dalla posizione del carattere e sia quelli appartenenti a sottolivelli dello stesso. Ad esempio "A.>" identifica i canali "A.B" e "A.B.C" ma non i canali "B.C" o "A".

2.4.2.2 Pattern di messaggistica supportati

Come detto in precedenza, KubeMQ supporta tutti i maggiori pattern di comunicazione. Si parte dal comune pattern della coda classica di messaggi, passando per il meccanismo *Publisher/Subscriber* per poi giungere al modello RPC.

Il primo pattern di comunicazione supportato è quello della coda di messaggi, definito, nella documentazione di KubeMQ come *Message Queues*. Tale meccanismo ha come oggetto cardine il messaggio. Esso viene inviato da un determinato mittente e preso in carico dal broker che lo conserva all'interno di una coda FIFO. Il messaggio rimane all'interno di questa coda fino a quando un destinatario consulta la coda stessa al fine di consumare il messaggio. Esistono dei limiti in quanto a dimensione del messaggio. Il pattern base è poi arricchito con numerose caratteristiche tipiche dell'ambiente KubeMQ.

Il sistema di messagistica a coda di KubeMQ fornisce la garanzia di consegna del messaggio attraverso quello che è il modello *at least once*[27], sebbene venga specificato che nella maggior parte dei casi il messaggio è consegnato esattamente una volta. È inoltre possibile inviare/ricevere più messaggi attraverso un'unica interazione con il broker. I messaggi possono essere marchiati attraverso una data di scadenza oltre la quale finiscono all'interno di una coda di supporto chiamata *dead-letter*. I destinatari possono mettersi in attesa di ricevere i messaggi in coda, leggerli senza consumarli e prenderli in carico impostando un tempo in cui assumono la visibilità esclusiva del messaggio stesso; possono inoltre rinviare un messaggio appena consumato o su un'altra cosa oppure modificandolo sullo stesso *channel*.

Il pattern *Pub/Sub* pone il focus sul concetto di evento, ben diverso dal concetto di messaggio analizzato in precedenza. Se il messaggio infatti, presuppone una conoscenza, da parte del mittente, riguardo al destinatario a cui esso deve fare riferimento, l'evento non comprende questa caratteristica. Ne deriva dunque una profonda differenza

[27] Il modello di consegna at least once garantisce la consegna effettiva del messaggio almeno una volta.

concettuale, la quale poi si traduce in differenze sia dal punto di vista implementativo e sia dal punto di vista degli use-case supportati.

In generale, tale pattern consiste nell'avere due entità: il Publisher, responsabile di pubblicare eventi in maniera del tutto asincrona su determinati *channel*, e il Subscriber, responsabile di iscriversi ai *channel* di interesse al fine di ricevere le informazioni dal Publisher. Dal punto di vista implementativo, ogni qualvolta che un Subscriber decide di iscriversi ad un *channel*, esso viene aggiunto ad una lista, collegata a quel *channel*. Nel momento in cui un Publisher decide di pubblicare un evento, esso viene inviato a tutti i Subscriber presenti nella lista relativa al *channel*, nel caso in cui questa lista si trovasse ad essere vuota, nessun *Subscriber* riceverebbe l'evento.

KubeMQ arricchisce il pattern classico con numerose implementazioni. In KubeMQ, infatti, esiste una distinzione tra il concetto di *event* ed *event store*. Il primo rispecchia in maniera abbastanza fedele il meccanismo analizzato in precedenza, il secondo, invece, consiste nell'avere un pattern Pub/Sub con persistenza degli eventi. La conseguenza di tale approccio si traduce nella possibilità di poter richiedere, nel momento dell'iscrizione, eventi pubblicati in precedenza.

Altra interessante feature per quanto riguarda il pattern Pub/Sub, presente in KubeMQ, riguarda il *grouping*. Attraverso di esso è possibile aggiungere competizione tra i vari Subscriber, in quanto solo uno di essi, appartenenti allo stesso gruppo, riceverà l'evento sul *channel* per cui è in ascolto. Sarà compito del broker comportarsi da load balancer, in modo da bilanciare la ricezione dei diversi Subscriber.

L'ultimo pattern implementato da KubeMQ che rimane da analizzare è quello relativo all'RPC, Remote Procedure Call. L'RPC è un protocollo che permette di trasferire il concetto di "chiamata a funzione" dall'ambiente locale a quello distribuito. Il tutto funziona attraverso la gestione di middleware incaricati di rilevare il Server e gestire il passaggio di parametri e valori di ritorno. Tale protocollo è implementato ed arricchito da KubeMQ. Il broker nativo di Kubernetes diversifica *Query* da *Command*. Le due tipologie differiscono semplicemente in base al valore di ritorno, se il *Command* può fornire come unici valori di ritorno quelli relativi alla riuscita con o senza errori del comando richiesto, le *Query* possono ritornare dei dati al processo chiamante. È poi possibile impostare un valore di timeout oltre il quale la procedura viene dichiarata fallita,

e si può, nella stessa maniera analizzata con il pattern Pub/Sub, utilizzare il *grouping* per impostare una sorta di load balancing. RPC per KubeMQ supporta infine il meccanismo di caching in modo da poter alleggerire il carico computazionale richiesto da più esecuzioni della stessa procedura.

2.4.3 KubeMQ vs Apache Kafka

In questo paragrafo verranno analizzate le differenze tra KubeMQ e uno dei broker più conosciuti ed apprezzati nell'era antecedente a Kubernetes, Apache Kafka. Prima di confrontare le due tecnologie, è opportuno spendere due parole su Apache Kafka.

Come appena detto, il prodotto di Apache è tra quelli più noti nell'ambito della messaggistica tra diversi microservizi. Esso è logicamente organizzato in *topics*, ognuno dei quali può avere zero o più produttori e consumatori. Gli eventi vengono pubblicati in questi *topic*, dai quali possono essere letti ogni volta che ce n'è bisogno. Il sistema è transazionale, ciò vuol dire che qualsiasi operazione avviene attraverso transazioni, dunque blocchi atomici. Un *topic* è concettualmente immaginabile come una categoria di messaggio, in maniera simile a quello che rappresenta un *channel* per KubeMQ. Essi sono organizzati come un file di log, e il consumatore è responsabile di tenere traccia della sua posizione corrente all'interno del file. Ogni *topic* si può dividere in partizioni.

Chiarito il funzionamento di base di Apache Kafka, si può passare a quello che è il confronto effettivo con KubeMQ. Tale confronto viene eseguito sulla base di diversi criteri [10]:

- Deployment: KubeMQ risulta essere estremamente semplice sia da comprendere che da configurare. Non richiede dunque la presenza di esperti specifici né la creazione di componenti di supporto. Kafka invece, presenta evidenti limiti se utilizzato in ambiente Cloud; ad esempio, la sua configurazione risulta essere abbastanza complessa, soprattutto se si prova ad installarlo in un sistema Kubernetes.
- Dimensioni: la dimensione di ogni container KubeMQ arriva ad un massimo di circa 30MB. Per quanto riguarda Kafka invece, esso è caratterizzato da una

dimensione di container di circa 600MB; per di più è necessario utilizzare un componente di supporto, *zookeeper*[28], il cui peso è di circa 100MB.

- Velocità: Essendo stato scritto utilizzando il linguaggio Golang, KubeMQ impiega circa il 20% in meno per gestire un milione di messaggi di dimensione di 1KB rispetto al suo avversario, il quale è stato sviluppato con Scala e Java.
- Pattern supportati: come visto nel paragrafo precedente, KubeMQ supporta code di messaggi, RPC e Pub/Sub con o senza persistenza. Kafka, invece, supporta solo il pattern Pub/Sub.

2.5 NodeJS

Altra tecnologia utilizzata nell'ambito del progetto in esame è rappresentata da NodeJS. NodeJS è un execution engine open-source e multipiattaforma per la realizzazione di programmi JavaScript. Esso non fornisce nessuna GUI e, a differenza di quanto avviene in ambito browser, non è eseguito in sandbox[29], dunque, permette l'esecuzione di programmi capaci di interagire direttamente con il sistema operativo, accedere a librerie native ed eseguire IPC[30] e comunicazioni di rete.

Figura 2.8: Logo NodeJS

NodeJS è utilizzato numerosi scopi differenti, tra cui ricordiamo:

- Nella creazione di server web;
- Per supportare il processo di sviluppo di un software;
- Per creare utilities
- Per interagire con DBMS

Esso risulta essere estremamente semplice da comprendere, efficiente e scalabile, si basa infatti su un insieme di nodi, da cui il nome "NodeJS", che vengono gestiti da un load balancer.

---

[28] Nelle ultime versioni Apache Kafka funziona anche senza l'utilizzo di Zookeeper [12].
[29] Un ambiente sandbox è un ambiente isolato, ai fini di ottenere maggiore sicurezza, dal resto dell'elaboratore
[30] IPC sta per Inter Process Communication e si riferisce ai meccanismi di comunicazione tra diversi processi

Si passa ora ad analizzare quella che è l'architettura classica di un programma NodeJS. Quando esso viene lanciato, tramite il comando "node foo.js", le istruzioni contenute al suo interno vengono eseguite nell'ordine dato. Se un'istruzione è costruita per fornire un risultato in futuro, NodeJS istanzia una coda di messaggi ed aspetta il messaggio indicante la fine di tale istruzione. All'arrivo di questo messaggio, viene lanciata la corrispondente callback. Quando tutte le istruzioni sono state eseguite e non ci sono messaggi pendenti nella cosa, NodeJS termina l'esecuzione.

NodeJS è stato progettato per supportare al massimo la modularità del codice, affinché le varie funzionalità di un programma possano rimanere disgiunte da un punto di vista logico. L'implementazione di funzionalità separate è dunque implementata in file diversi al fine di rendere questi moduli indipendenti e riusabili. Altro vantaggio derivante dalla divisione del codice in moduli, si può trovare nell'aumento della leggibilità e della comprensibilità del codice.

NodeJS supporta inoltre, sia l'approccio legato a CommonJS, basato sulla funzione *require(...)*, e sia quello basato su ES6, basato sullo statement *import*. Il primo fu introdotto all'inizio del progetto ed è capace di localizzare sia moduli *built-in*, sia moduli di terze parti piuttosto che moduli locali. Il secondo invece richiede la specifica nel file chiamato "package.json" della entry "type":"module", ed è disponibile a partire dalla versione 15.3.0. Molto spesso i moduli non sono creati dal team che sta sviluppando il progetto, ma sono forniti dalla community. Nello specifico il programma npm, NodeJS Package Manager, consente di localizzare, scaricare e installare moduli di terze parti all'interno del progetto che si sta sviluppando.

Come si può osservare dalla Figura 2.8, le prestazioni comparate con altri framework nel caso in cui si voglia ottenere un servizio che fornisca una lista complessa di tutti gli utenti che hanno visitato la Francia, mostrando anche tutti i paesi visitati, danno la netta consapevolezza di quanto NodeJS sia performante, soprattutto nel caso in cui si utilizzi una macchina fornita di 12 core e 32 GB di RAM, dove quello con NodeJS risulta essere il sistema più performante con uno scarto rispetto al secondo di questa classifica, PHP Lumen, di circa 200 richieste gestite al secondo [11].

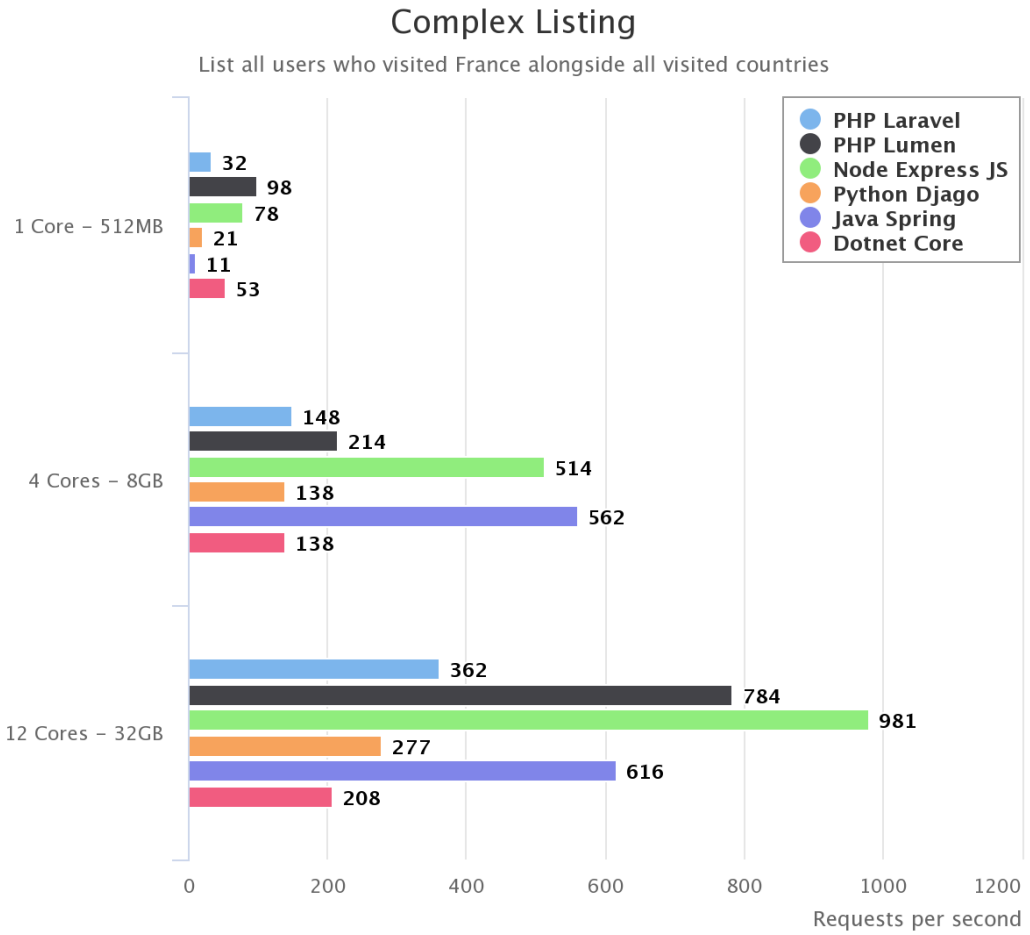


Figura 2.9: Comparazione delle performance tra diversi framework[31]

2.6 KubeVirt

Sebbene non sia stata poi effettivamente utilizzata all'interno della soluzione proposta, merita comunque menzione la tecnologia relativa a KubeVirt. KubeVirt è un progetto, appartenente al CNCF Landscape, nato dall'esigenza di poter utilizzare il concetto di macchina virtuale all'interno di un cluster Kubernetes. Sebbene, come indicato nel paragrafo 2.2, nella sezione relativa alle ere della distribuzione dei servizi, esistono casi in cui:

[31] Fonte: https://medium.com/@mihaigeorge.c/web-rest-api-benchmark-on-a-real-life-application-ebb743a5d7a3

- Non è semplice andare a rendere containerizzare[32] una specifica applicazione o carico di lavoro;
- Ci si trova nel caso in cui è comunque ancora necessario l'isolamento fornito dalle virtual machine.

KubeVirt si sviluppa, dunque, proprio da queste necessità, per permettere alle macchine virtuali di poter essere utilizzate nell'ambito di Kubernetes e di poter interagire con i pod.

Dal punto di vista tecnico, installato KubeVirt all'interno del proprio cluster, esistono diversi metodi per avviare una virtual machine. È possibile, infatti avviare direttamente un'istanza di VM, piuttosto che creare un oggetto *VirtualMachine* da accendere e spegnere nei momenti in cui se ne presenta la necessità. È possibile collegare supporti esterni, quali ad esempio risorse di archiviazione, ed eseguire attraverso esse anche configurazioni delle macchine per mezzo di *cloud-init*[33].

Per quanto concerne l'utilizzo delle immagini dalle quali partire, anche in questo caso, è possibile utilizzare e far partire macchine con diversi meccanismi:

- È possibile andare a specificare all'interno del manifest, relativo alla creazione della risorsa *VirtualMachine*, quale sia l'immagine da utilizzare. Essa può essere un'immagine live o ISO, dalla quale fare partire l'installazione del sistema operativo.
- È possibile creare dei *Persistent Volume Claim*, sui quali andare ad installare un sistema operativo, per poi collegarlo come disco alla VM, al fine di ottenere anche persistenza. È poi possibile clonare questi PVC in modo da ottenere degli snapshot o delle operazioni di clonaggio delle macchine stesse.

Ad ogni modo, le immagini vengono scaricate e convertite nel formato corretto attraverso un componente, anch'esso installabile all'interno del cluster, chiamato *Containerized Data Importer*, *CDI*. Tale componente supporta il download e la conversione di immagini, per la corretta esecuzione della VM, a partire da file di tipo "raw" e "qcow2" anche compressi nei formati "gz" e "xz". CDI permette di scaricare immagini sia da URL

[32] Trasformare in container.
[33] Cloud-init è un meccanismo standardizzato attraverso il quale è possibile eseguire la prima configurazione di un sistema operativo Linux.

e sia da Registry. Nel secondo caso l'immagine deve essere impacchettata in una Docker Image della quale verrà automaticamente eseguito il pull [12].

Capitolo Terzo

Capitolo 3

3 Architettura ed evoluzione

Analizzate alcune delle principali tecnologie affrontate nel corso dello sviluppo della soluzione, oggetto di questo capitolo, è l'analisi dell'architettura proposta, l'evoluzione della stessa, dettata dai limiti riscontrati, e l'implementazione di un buon meccanismo di monitoring.

3.1 Container Windows

Fin da subito è sembrato chiaro che la direzione da seguire, al fine di poter raggiungere gli obiettivi prefissati, fosse rappresentata dall'utilizzo di Kubernetes. Il primo passo concreto, nell'ambito dell'implementazione della soluzione, è stato dunque quello di andare ad inserire, all'interno di un container, l'ambiente di rendering. Tale ambiente prevede:

- Un sistema operativo Windows, in quanto, Adobe After Effects non risulta essere installabile su sistemi operativi Linux;
- L'installazione effettiva del tool di Adobe;
- L'installazione dell'ambiente di NodeJS al fine di poter inserire della logica e dei meccanismi di interazione con una coda.

Il focus primario è stato dunque puntato sullo studio dei container windows, sul loro supporto e sui loro limiti. Tali container possono essere eseguiti solo su host con Sistema Operativo Windows [13], con una build superiore o uguale rispetto a quella dell'immagine base del container stesso. È ad esempio impossibile eseguire un container con immagine base *Windows 10 versione 20H2* su un host con sistema operativo *Windows 10 versione 2004*.

Altra limitazione caratteristica dei container Windows è il tempo di avvio degli stessi, superiore di un ordine di grandezza rispetto al corrispettivo Linux. Se un container

Ubuntu, ad esempio, impiega circa 1 secondo per essere operativo, un container Windows 10, ne impiega circa 12.

Si passa poi ad analizzare quella che è la dimensione di un container Windows. Esso, in generale, ha una dimensione, decisamente maggiore rispetto a quella di un container Linux. Ciò che risulta, quindi, dallo studio della tecnologia riguardante i container Windows, è che essi sono globalmente più complessi da utilizzare, rispetto ai più classici container Linux.

È inoltre importante sottolineare che, al fine di poter utilizzare Adobe After Effects, è necessario che l'immagine base dei container abbia un ambiente grafico. Non è, ad ora, dunque consentito l'utilizzo di immagini *Windows Nano Server* e *Windows Server Core*, in quanto mancanti di DLL[34] grafici necessari affinché l'installazione del tool di Adobe vada a buon fine.

3.1.1 Alternative ai container Windows

Alla luce di queste considerazioni, è stato fatto il possibile affinché potesse essere evitato l'utilizzo di container Windows. Tuttavia, questi tentativi non hanno portato al risultato sperato, e dunque, nella soluzione proposta, sono stati poi utilizzati tali container. Si analizzano ora quelle che sono state le alternative testate e i limiti delle stesse che hanno portato al loro abbandono.

La prima alternativa analizzata consiste nell'utilizzo di tool, quali ad esempio WineHQ, capaci di eseguire programmi Windows su sistemi operativi Linux. Questo non avviene tramite tecniche di emulazione, da cui il nome del software analizzato, *Wine Is Not an Emulator* [14]; ma attraverso uno strato di compatibilità introdotto dal tool stesso al fine di tradurre le API Windows in chiamate alle API dei sistemi operativi su cui esso viene eseguito

Sebbene si sia riusciti, dopo numerosi tentativi, attraverso tale tool, ad installare Adobe After Effects su Linux, il programma non si avvia a causa della mancanza di determinati DLL. Questa problematica sembrerebbe essere risolvibile ma evidenzia il limite

[34] Un DLL, Dynamic Link Library, è una libreria con codice eseguibile, che può essere utilizzata contemporaneamente da più processi. È tipica dei sistemi operativi prodotti da Microsoft

intrinseco della soluzione in analisi. L'utilizzo di software come Wine non rappresenta una soluzione definitiva, in quanto si tratta di un adattamento instabile. Non è possibile ottenere la garanzia che un futuro aggiornamento rilasciato da Adobe, non vada a richiedere nuovi DLL, andando a rompere il sistema creato per versioni precedenti.

Si è dunque deciso di provare ad evitare l'utilizzo dei container Windows, esplorando la soluzione KubeVirt, analizzata nel paragrafo 2.6. Tuttavia, sebbene promettente, essa presenta ancora diversi limiti. L'idea alla base di questa esplorazione era quella di andare ad utilizzare delle VM Windows, piuttosto che dei container, al fine di poter garantire un utilizzo migliore delle risorse. Il vantaggio di questa soluzione è rappresentato dal fatto che le VM Windows possono essere eseguite anche su nodi Linux, a differenza dei container per cui non vale questo discorso. Tale strada è stata poi abbandonata a causa di diverse problematiche, tra cui:

- Poca chiarezza sull'utilizzo delle licenze Windows, in ambito VM, per cui non è stato possibile effettuare test approfonditi;
- Impossibilità di condivisione dei volumi, necessaria ai fini del progetto, tra più VM;
- Tempi di avvio ancora maggiori rispetto a quelli misurati nell'utilizzo di un container.

Si è dunque deciso, alla luce delle problematiche emerse durante la fase di studio di alternative, di utilizzare i container Windows, e nello specifico, *Windows 10 versione 1809*, sebbene non sia la soluzione ottimale. Sarà noto il motivo di tale decisione nei paragrafi successivi.

3.2 Architettura

Avendo compreso quali siano le problematiche e i limiti che hanno portato alla scelta di utilizzare i Windows Container, si passa ora ad analizzare quella che è l'architettura proposta e come essa è stata adattata a funzionare su Azure Kubernetes Service.

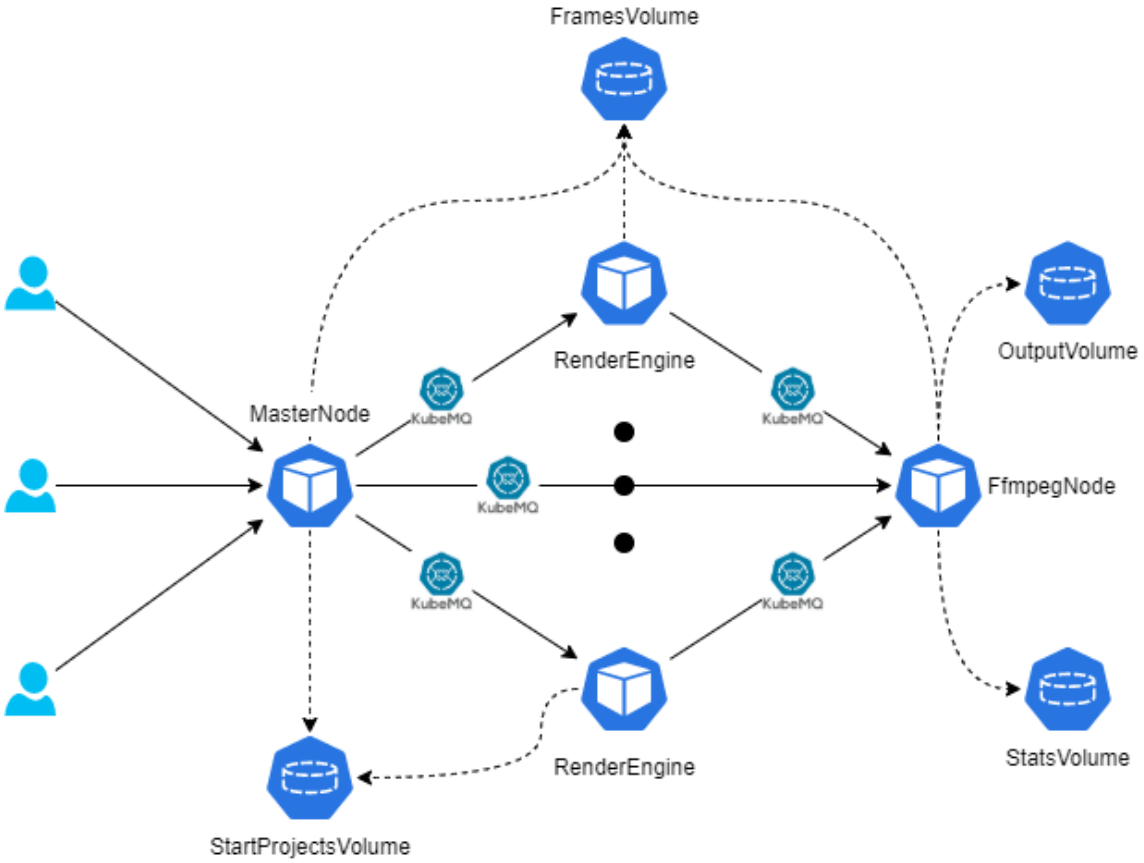


Figura 3.1: Architettura proposta per realizzazione sistema di rendering in Kubernetes

Come si evince dalla Figura 3.1, l'architettura per la realizzazione del sistema consiste in una serie di componenti che si interfacciano tra di loro tramite meccanismi appositamente costruiti. Sostanzialmente il sistema si costituisce di tre blocchi, pod, fondamentali, che interagiscono tra loro tramite KubeMQ e volumi condivisi, essi sono stati denominati:

- MasterNode;
- RenderEngine;
- FfmpegNode.

L'architettura in esame ha l'obiettivo di sfruttare i vantaggi forniti dalla massima parallelizzazione possibile del processo di rendering. In breve, ogni progetto viene suddiviso in parti uguali dal MasterNode, ogni RenderEngine effettua poi il rendering della porzione del progetto che gli è stata assegnata e, successivamente, FfmpegNode assembla le varie porzioni del progetto di cui è stato effettuato il rendering per costruire il video finale.

Sarà oggetto dei seguenti paragrafi quello di analizzare ogni componente, le sue peculiarità, i limiti e le scelte implementative.

3.2.1 MasterNode

Quello definito in Figura 3.1 come *MasterNode* è il nodo, all'interno dell'architettura, che ha il compito di interfacciarsi con l'utente. Esso è costituito da un pod, in cui viene eseguito un container che espone due endpoint HTTP attraverso i quali è possibile avviare il processo di rendering. Nello specifico si tratta degli endpoint "*/renderPNG*" e "*renderAVI*", accessibili tramite richieste POST, passando tramite body un oggetto JSON contenente informazioni riguardo al progetto per cui effettuare il rendering e alle diverse opzioni relative al processo.

La discriminante sulla scelta dell'endpoint risiede nella modalità con cui viene poi effettuato il rendering. È possibile avviare un rendering in modalità PNG, per cui il workflow consiste nella realizzazione delle varie porzioni attraverso la costruzione dei file, in PNG appunto, di ogni frame appartenente alla parte del video indicata. Tali frame verrebbero poi passati in pipe e concatenati dall'*ffmpegnode*. Nel caso in cui invece si dovesse scegliere la modalità AVI, il rendering delle varie porzioni avverrebbe nell'omonimo formato, e i vari spezzoni sarebbero poi concatenati dall'*ffmpegnode*. Dalla ricerca empirica, realizzata attraverso numerosi test, è emerso che, tendenzialmente per video brevi, con un numero massimo di circa 600 frames, la modalità AVI risulta essere più veloce, mentre per video lunghi la modalità PNG sembra essere più vantaggiosa. Questo è dovuto al fatto che, sebbene la procedura di realizzazione dei PNG risulti essere più lenta, il tempo perso durante questa fase viene recuperato dal fatto che l'istanza di *ffmpeg* si avvia fin da subito e si rende pronta a ricevere i frame in pipe. Contrariamente a quanto avviene per la modalità AVI in cui, sebbene la fase di rendering della porzione indicata sia più rapida, l'istanza di *ffmpeg* può partire solo quando tutte le porzioni sono state realizzate.

Alla ricezione di richieste su tale endpoint, il pod reagisce seguendo questa procedura:

1) Viene effettuato un controllo atto a garantire l'effettiva esistenza del progetto per cui si vuole avviare il rendering, e, nel caso in cui esso non dovesse esistere, verrebbe interrotta la procedura e si risponderebbe al client con uno specifico

messaggio d'errore. Il progetto, nel momento in cui viene effettuata la richiesta, deve essere presente all'interno del volume *StartProjectsVolume*, le modalità con cui il trasferimento dei file all'interno di tale volume avviene, non è oggetto del progetto in analisi.

2) Viene creata, all'interno del volume *FramesVolume*, una cartella, relativa al progetto, nella quale saranno salvati i file utili al processo. Nel caso in cui la cartella dovesse già essere presente, e dunque dovesse essere già in corso un'operazione di rendering per il progetto richiesto, si risponderebbe al client con uno specifico messaggio d'errore.
3) Si passa poi ad una fase di divisione del lavoro. In questa fase, il *MasterNode* effettua un controllo per ottenere l'informazione riguardante il numero delle istanze attive di *RenderEngine*. Nel caso in cui, per qualsiasi motivo, il *MasterNode*, non dovesse riuscire a ricavare tale informazione, il progetto verrebbe suddiviso in base ad un fattore di divisione, indicabile, all'interno del manifest del deployment relativo al *MasterNode*, tramite variabile d'ambiente. In base al numero ottenuto, si divide il progetto in maniera pressoché equa, tra le diverse istanze di rendering. Ogni progetto viene quindi scomposto in "N" blocchi, con "N" uguale al numero delle istanze di rendering attive.
4) Vengono informati, a questo punto, tutti gli attori coinvolti nel processo di rendering attraverso le code KubeMQ. Le informazioni relative alla divisione del lavoro vengono comunicate attraverso il canale *renderengine*.
5) Nel frattempo, viene anche inviato un messaggio sul canale *ffmpegnode* affinché possa essere predisposta un'istanza di *FfmpegNode* che possa gestire il processo di rendering richiesto.

Il meccanismo analizzato nelle righe precedenti è realizzato attraverso lo sviluppo di un'applicazione NodeJS, capace di ricevere richieste su endpoint HTTP, grazie all'utilizzo del noto framework *express.js*, e incaricata di gestire i meccanismi relativi alle code KubeMQ e all'interazione con Kubernetes attraverso le loro librerie ufficiali.

È possibile personalizzare il comportamento delle istanze di MasterNode grazie all'impostazione di opportune variabili d'ambiente. Nella Tabella 2, tali variabili vengono presentate.

Tabella 2: Variabili d'ambiente MasterNode

Nome Variabile	**Utilizzo**
SERVER_PORT	Porta di ascolto del server.
DEFAULT_SPLITTING_INDEX	Fattore di divisione dei progetti, nel caso in cui non si riesca a reperire il numero di istanze di RenderEngine attive.
RENDER_ENGINE_DEPLOYMENT	Nome del deployment dei RenderEngine.
DEFAULT_CHANNEL_RENDER_ENGINE	Nome del canale di comunicazione con i RenderEngine.
KUBEMQ_ADDRESS	Indirizzo del broker KubeMQ.
DEFAULT_CHANNEL_FFMPEGNODE	Nome del canale di comunicazione con FfmpegNode.
LOGGING_LEVEL	Livello di logging.
OUTPUT_DIRFRAMES_BASE_PATH	Cartella nella quale devono essere salvati i file intermedi dei progetti.

3.2.2 RenderEngine

Il *RenderEngine* è il componente principale del sistema ideato. Esso ha infatti il compito di andare ad effettuare in maniera concreta il rendering delle porzioni, per la costituzione del video. Anch'esso, così come il MasterNode, si basa su un algoritmo ben preciso, messo in atto grazie alla costruzione di un'applicazione in NodeJS. Tale algoritmo si struttura nei seguenti step:

1) Fin dalla partenza, il processo si mette in ascolto, sul channel *renderengine*, pronto a ricevere messaggi dal MasterNode. Tali messaggi prevedono l'esistenza di quattro campi fondamentali, i quali indicano il frame di partenza, il frame di conclusione, la modalità di rendering selezionata e il progetto su cui si deve lavorare. Tale progetto, così come descritto nell'analisi dell'algoritmo caratterizzante il MasterNode, deve essere presente all'interno del volume *StartProjectsVolume*.
2) Alla ricezione di un nuovo messaggio, il sistema reagisce andando ad avviare la procedura di rendering della porzione di video indicata. Tale processo, porta alla creazione dei file relativi al blocco desiderato, in formato PNG, nel caso in cui si sia scelta tale modalità, o in formato AVI.

3) Ogni volta che viene ricevuto un messaggio, questo viene preso in carico e ne viene tolta la visibilità agli altri RenderEngine in ascolto sullo stesso canale. L'esclusiva visibilità del messaggio ha una durata prestabilita, indicabile tramite variabili d'ambiente, ed è rinnovata periodicamente durante l'effettiva operazione di rendering. Se qualcosa, durante questa fase, dovesse non funzionare, la visibilità esclusiva non verrebbe rinnovata e il messaggio tornerebbe in coda per essere gestito da altre istanze di RenderEngine.
4) I file risultanti dal processo di rendering vengono salvati all'interno di una cartella indicata tramite variabile d'ambiente, all'interno del volume *FramesVolume.* Il nome dei fotogrammi all'interno di tale cartella risulta essere "*frame[####].png*", dove con "#" si indica il numero, su quattro cifre, del fotogramma all'interno del video, mentre il nome delle porzioni AVI è "*video#.avi*" dove "#" indica il numero del primo frame del blocco.
5) Quando il rendering termina, viene eseguito il cosiddetto *ack* del messaggio relativo, procedura attraverso la quale si va ad eliminare completamente il messaggio dalla coda, in quanto gestito, e l'algoritmo riparte dallo step 1.

Il comportamento del RenderEngine è personalizzabile attraverso l'utilizzo di variabili d'ambiente. Esse sono illustrate nella Tabella 3.

Tabella 3: Variabili d'ambiente RenderEngine

Nome Variabile	**Utilizzo**
KUBEMQ_ADDRESS	Indirizzo del broker KubeMQ.
DEFAULT_CHANNEL_RENDERENGINE	Nome del canale della coda da cui vengono ricevuti i messaggi dal MasterNode.
VISIBILITY_SECONDS	Periodo di uso esclusivo del messaggio espresso in secondi.
WAIT_TIMEOUT_SECONDS	Periodo di attesa di un messaggio sulla coda.
OUTPUT_BASE_PATH	Percorso in cui devono essere salvati i vari file di output.
INPUT_BASE_PATH	Percorso dal quale devono essere caricati i progetti da elaborare
AERENDER_PATH	Percorso del tool aerender.exe.
LOGGING_LEVEL	Livello di logging.

Aspetto fondamentale da trattare, per quanto riguarda il RenderEngine, è l'immagine base. Sebbene da un punto di vista logico, sia ottimale l'utilizzo di un container che abbia come immagine base una versione di Windows Server, capace di sfruttare al meglio le accelerazioni hardware, e capace dunque di fornire prestazioni migliori, attualmente tale scenario non risulta essere praticabile sul servizio Kubernetes di Azure.

Come detto nel paragrafo relativo ai container Windows, il 3.1, tali container possono essere eseguiti solo su host che abbiano una versione superiore o uguale rispetto a quella dell'immagine base del container stesso. Questo è un aspetto fondamentale nella scelta dell'immagine base per quanto riguarda il RenderEngine, in quanto è stata scelta l'immagine di *Windows 10 versione 1804*. Tale scelta è stata fatta a causa del fatto che attualmente, sui nodi Windows all'interno di Azure Kubernetes Service, è installato il sistema operativo *Windows Server 2019 versione 1804*. Dunque, non è possibile eseguire, su AKS container che abbiano immagini base di versione superiore alla 1804. Microsoft però nel suo repository in Docker Hub, relativo alle immagini base Windows, non ha rilasciato alcuna immagine base Windows Server di versione inferiore alla 1804 che fornisca un ambiente desktop, necessario, come detto in precedenza, per l'installazione di Adobe After Effects. Ad oggi, l'unica immagine server con ambiente desktop pubblicata da Microsoft è quella di Windows 2022, dunque non utilizzabile sui nodi Windows di AKS. Per questo motivo, si è deciso di utilizzare un'immagine non server come base, e, nello specifico la 1804 in quanto l'unica utilizzabile tra quelle presenti in Docker Hub. È tuttavia stata annunciata l'intenzione, da parte di Microsoft, di aggiornare le immagini delle VM per AKS con l'installazione del sistema operativo Windows 2022. Tale aggiornamento, nel momento in cui sarà realizzato, contribuirà ad aumentare in maniera netta le performance del sistema.

Aspetto interessante riguardo allo sviluppo della soluzione è quello concerne il meccanismo di terminazione del processo di *aerender*. All'interno di un container, tale procedura, alla fine del processo di rendering, richiede infatti un periodo di tempo non trascurabile. Tale tempo di terminazione non è riscontrabile invece, eseguendo *aerender* come normale processo direttamente sull'host. Al fine di ridurre, dunque, questa tempistica e migliorare quindi le performance, il processo viene terminato in maniera forzata nel momento in cui ha effettivamente terminato il suo lavoro, in modo tale da permettere al pod di *RenderEngine* di tornare in ascolto di nuovi messaggi.

3.2.3 FfmpegNode

Il componente presentato nel presente paragrafo è *FfmpegNode*, nodo responsabile della creazione effettiva del video, per mezzo dell'assemblaggio, tramite *ffmpeg*[35] delle porzioni create dai RenderEngine. Esso implementa un algoritmo realizzato attraverso l'esecuzione di un'applicazione NodeJS. Tale algoritmo consiste nei seguenti passaggi:

1) Fin dal suo avvio, l'FfmpegNode, si mette in ascolto sul canale *ffmpegnode*, in attesa di ricevere messaggi.
2) Alla ricezione di ogni messaggio il processo reagisce in maniera analoga a quanto avviene per i RenderEngine, si assicura dunque la visibilità esclusiva del messaggio per un periodo di tempo prestabilito ed indicabile tramite variabili d'ambiente, e rinnova tale esclusiva periodicamente fino a che non si verificano errori, in tal caso, l'esclusiva non verrebbe rinnovata ed il messaggio finirebbe nuovamente nella coda, pronto ad essere gestito da altre istanze.
3) Successivamente viene fatta partire un'istanza di *ffmpeg*, la quale viene avviata seguendo quelle che sono le opzioni di rendering indicate nella richiesta iniziale al MasterNode e passate all'FfmpegNode attraverso la coda KubeMQ. Tale istanza di ffmpeg si prepara dunque ad assemblare fotogrammi ricevuti in pipe tramite lo *stdin* oppure ad assemblare le porzioni di video in AVI.
4) A questo punto, ci si mette in attesa dei frame o AVI, sul percorso indicato, all'interno del volume *FramesVolume*, e, attraverso un loop, si controlla l'effettiva presenza dei file e il passaggio degli stessi all'istanza di *ffmpeg*.
5) Quando l'operazione di *ffmpeg* termina, viene effettuato l'*ack* del messaggio relativo, viene cancellata la cartella dei file di input all'interno del volume *FramesVolume*, ed il video risultante, viene salvato all'interno del volume *OutputVolume*.
6) Al termine di tutte le operazioni, l'algoritmo riprende dallo step 1.

Altro aspetto da considerare, per quanto concerne l'FfmpegNode, è quello relativo alla sua connessione al volume *StatsVolume*. Tale volume è utile al fine di costruire statistiche di funzionamento in modo da poter analizzare le performance del sistema. Nello

[35] Ffmpeg è un tool che permette di lavorare sulla creazione di video tramite linea di comando.

specifico, ogni volta che termina correttamente un'istanza di *ffmpeg* viene accodata, ad un file in formato *csv*[36], una riga contenente, tra altre informazioni relative al progetto, la durata dell'intero processo di rendering.

Come per gli altri componenti analizzati, anche in questo caso è possibile personalizzare il comportamento dell'FfmpegNode tramite l'utilizzo delle variabili d'ambiente. Tali variabili sono illustrate nella Tabella 4.

Tabella 4: Variabili d'ambiente FfmpegNode

Nome Variabile	**Utilizzo**
KUBEMQ_ADDRESS	Indirizzo del broker KubeMQ.
DEFAULT_CHANNEL_FFMPEGNODE	Nome del canale della coda da cui si ricevono i messaggi da MasterNode.
VISIBILITY_SECONDS	Periodo di uso esclusivo del messaggio.
WAIT_TIMEOUT_SECONDS	Periodo di attesa di un messaggio su coda.
OUTPUT_BASE_PATH	Percorso in cui verranno salvati i video elaborati.
INPUT_BASE_PATH	Percorso da cui verranno presi i file da concatenare per la costruzione del video.
PATH_STATS	Percorso in cui sono salvati i csv relativi ai tempi di elaborazione.
LOGGING_LEVEL	Livello di logging.

Aspetto interessante che riguarda lo sviluppo del componente in analisi è quello relativo alla sincronizzazione. L'interrogativo a cui si vuole rispondere in questa fase si riferisce al come, i *RenderEngine* e gli *FfmpegNode* si coordinino. Per quanto riguarda la modalità PNG, la sincronizzazione avviene grazie ad un normalissimo *fileWatcher* che rimane in attesa e controlla la ricezione dei frame. Questa soluzione, tuttavia, non è adatta nel caso in cui la modalità scelta sia AVI, in quanto, le dimensioni decisamente maggiori degli spezzoni di video, rispetto a quelle dei frame, provocano problemi nel funzionamento del *fileWatcher*. Dunque, per ovviare a questa problematica, sono stati introdotti dei *file di controllo*, i quali vengono creati dai *RenderEngine* nel momento in cui essi terminano la loro elaborazione. Il *fileWatcher* del pod *FfmpegNode* si metterà dunque in attesa di tali file.

[36] CSV sta per Comma Separated Values ed indica un formato di file di testo la cui formattazione prevede che i diversi valori siano separati da virgole.

3.2.4 Scaling

Illustrati quelli che sono i componenti fondamentali dell'architettura, si passa ora all'analisi del processo di scaling degli stessi, discutendo, in base alle analisi fatte nel paragrafo 1.4.1, l'effettiva utilità di un approccio reattivo, piuttosto che uno predittivo. Considerando che i dati relativi alle elaborazioni, sono avvenute in una fase finale del progetto, è stato comunque pensato un meccanismo che potesse permettere al sistema di scalare in maniera opportuna, reagendo al carico di lavoro richiesto.

Tale processo potrebbe essere reso possibile, per quanto riguarda il *MasterNode* attraverso l'utilizzo dell'*Horizontal Pod Autoscaling*, andando ad impostare in maniera opportuna, all'interno del manifest di definizione, le varie soglie utili a permettere l'operazione di ridimensionamento.

Per quanto riguarda i nodi di *RenderEngine* e gli *FfmpegNode* invece, è stato costruito un componente ad hoc, lo *ScalerNode*, il quale è costituito da un'applicazione NodeJS che esegue esattamente il seguente algoritmo:

1) Controlla periodicamente il numero di messaggi in coda sul canale indicato attraverso le variabili d'ambiente;
2) Inserisce tale numero all'interno di un vettore rappresentante la storia degli ultimi riscontri dal precedente test;
3) Calcola la media di tale vettore;
4) Se essa è superiore alla soglia massima indicata, viene fatta partire (e si attende la fine) la procedura di ridimensionamento al fine di aumentare le istanze del deployment indicato, sempre che queste non siano già uguali al numero massimo consentito;
5) Se essa è inferiore alla soglia minima indicata, viene fatta partire (e si attende la fine) la procedura di ridimensionamento al fine di diminuire le istanze del deployment indicato, sempre che queste non siano già uguali al numero minimo consentito.

È possibile notare che il comportamento del componente in esame è personalizzabile attraverso le variabili d'ambiente. Impostando opportunamente tali variabili risulta estremamente semplice andare a creare un'istanza di *ScalerNode* per la gestione dei

RenderEngine, piuttosto che una per la gestione degli *FfmpegNode*. È possibile visionare l'elenco delle variabili d'ambiente per lo *ScalerNode* attraverso la Tabella 5.

Tabella 5: Variabili d'ambiente ScalerNode

Nome Variabile	**Utilizzo**
DEPLOYMENT_TO_SCALE	Nome del deployment per cui effettuare lo scaling.
NS_DEPLOYMENT_TO_SCALE	Namespace del deployment per cui effettuare lo scaling.
DEFAULT_CHANNEL	Nome del canale da controllare.
MAX_HISTORY_SIZE	Dimensione del vettore rappresentante la storia delle misurazioni.
KUBEMQ_ADDRESS	Indirizzo del broker KubeMQ.
QUEUE_MAX_THRESHOLD	Soglia massima oltre la quale aumentare le istanze del deployment indicato.
QUEUE_MIN_THRESHOLD	Soglia minima sotto la quale diminuire le istanze del deployment indicato.
TIMER_CHECK_QUEUE_MILLIS	Intervallo di tempo tra due misurazioni, espresso in millisecondi
TIME_AFTER_SCALE	Intervallo di tempo da attendere ogni volta che si va ad avviare una procedura di ridimensionamento, al fine di controllare l'effettivo completamento.
MAX_PODS	Numero massimo di pod accettabili per il deployment indicato.
MIN_PODS	Numero minimo di pod accettabili per il deployment indicato.
MAX_RETRY	Numero massimo di tentativi, nel caso in cui qualcosa dovesse andare storto, prima di terminare il processo.
LOGGING_LEVEL	Livello di logging.

Avendo in mente come dovrebbe avvenire l'operazione di scaling dei nodi all'interno del cluster, si passa ora ad analizzare quello che è il supporto di Azure Kubernetes Service a questa operazione. Come già accennato nel paragrafo 2.3, la procedura di scaling, nel caso in cui le risorse del cluster dovessero essere sature, passa per una fase di ridimensionamento dei nodi. È stato misurato che questa fase, soprattutto per quanto riguarda i nodi windows, richiede del tempo, non trascurabile, prima che il nodo si accenda.

Altro aspetto fondamentale è dovuto al fatto che tali nodi devono poi scaricare l'immagine Docker al fine di avviare il container all'interno del pod. Questa operazione dipende strettamente dal numero di operazioni di I/O che la macchina scelta può effettuare al secondo, ma, tenendo conto della dimensione considerevole delle immagini, soprattutto quelle Windows, questo tempo non risulta mai essere trascurabile. Ad esempio, sono necessari complessivamente circa 15-20 minuti affinché possa essere correttamente ridimensionato il numero di *RenderEngine* se l'operazione richiede l'avvio di una nuova macchina con capacità di 12800 operazioni di I/O al secondo.

Tuttavia, Azure mette a disposizione delle immagini Docker precaricate all'interno delle VM Windows di AKS, proprio al fine di ridurre il tempo di *pull*, in quanto verrebbe eseguito poi il download dei soli layer non presenti. È possibile dunque utilizzare una delle immagini già presenti come immagine base per la realizzazione della *Docker Image* da eseguire nei vari pod. Il problema però nasce dal fatto che, attualmente, tra le immagini preconfezionate, nessuna di esse è compatibile con l'installazione di After Effects. Si rimane tuttavia fiduciosi riguardo al fatto che, con l'aggiornamento dei nodi a Windows Server 2022, saranno fornite immagini compatibili con l'installazione richiesta.

Alla luce delle analisi svolte nel primo capitolo, e considerando i limiti che attualmente esistono all'interno del meccanismo di Azure Kubernetes Service, si è deciso di abbandonare la strada relativa allo scaling reattivo. Rimane e si è analizzata comunque l'implementazione del componente *ScalerNode* per eventuali migliorie future in corrispondenza ad evoluzioni tecnologiche.

3.3 Monitoring

Un altro aspetto fondamentale, nella realizzazione di un sistema, è quello relativo al monitoring. Esso risulta essere estremamente importante per diverse ragioni. Sicuramente un buon meccanismo di monitoring permette di avere sempre a disposizione lo stato di salute del sistema, garantendo dunque di poter intervenire in maniera rapida e puntuale in caso di guasti. Non bisogna poi dimenticarsi della possibilità, fornita dal monitoring, di poter tracciare quelle che sono le performance, in relazione all'utilizzo delle risorse, permettendo di comprendere quali siano i vantaggi e i limiti delle diverse configurazioni, e consentendo dunque di sfruttare appieno l'infrastruttura a disposizione.

Nel progetto in esame, tale meccanismo è stato realizzato grazie all'utilizzo di un servizio offerto da Microsoft e denominato Azure Monitoring, il quale, grazie all'analisi di log, piuttosto che a continue interrogazioni sul server delle metriche del cluster Kubernetes, permette di costruire una dashboard facilmente accessibile dall'amministratore del sistema.

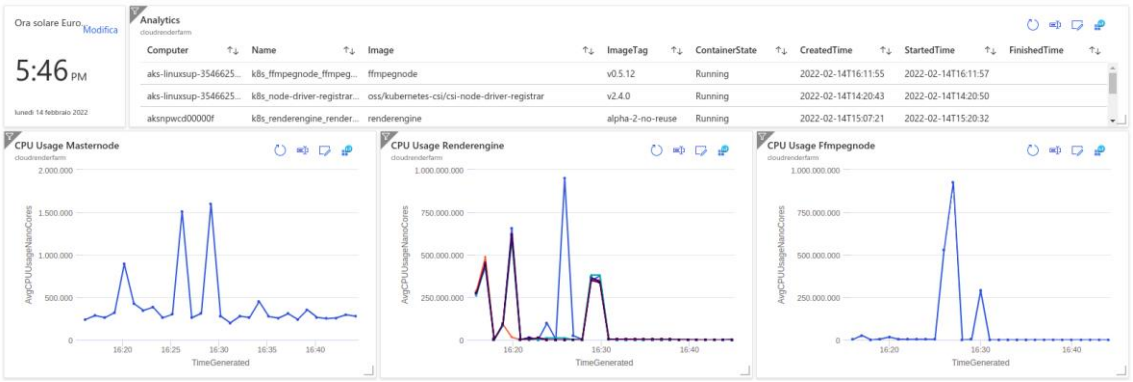


Figura 3.2: Screen della dashboard collegata al cluster Kubernetes

Nella Figura 3.2 è possibile osservare un estratto della dashboard di cui si è parlato sopra, nello specifico, in questa porzione è presente una tabella indicante lo stato di tutti i pod che sono in esecuzione all'interno del cluster, ed un resoconto su quello che è l'utilizzo in termini di CPU dei diversi componenti dell'architettura.

Capitolo Quarto

Capitolo 4

4 Costi e possibili configurazioni

Prima di concludere la trattazione dell'elaborato, è necessario fornire alcune informazioni riguardanti il costo di mantenimento dell'infrastruttura proposta nella presente tesi. Al fine di comprendere appieno questo aspetto, viene fornita in prima analisi una descrizione afferente a quali siano i costi in Azure Kubernetes Service e a come questi vengano gestiti. Segue poi una rapida illustrazione riguardante come gestire le diverse configurazioni al fine di comprendere come possa essere modellata la struttura della soluzione affinché rispecchi in maniera adeguata quelle che sono le esigenze del cliente.

4.1 Meccanismo costi Azure

Azure offre diversi servizi a pagamento, i quali sono utili nell'implementazione della soluzione analizzata. È possibile, grazie al servizio *Azure Pricing Calculator*, avere una stima dei costi da sostenere a partire da una configurazione data.

Aspetto particolare, ma ragionevole, riguarda il meccanismo di pagamento di Azure Kubernetes Service. Sebbene, infatti, il focus logico di un servizio Kubernetes sia posto sui pod, Azure concentra la sua dinamica di pagamento sull'utilizzo dei nodi. Da questa meccanica, risulta dunque palese che, il miglior modo per sfruttare appieno la potenza di calcolo per cui si paga, anche in Kubernetes, consiste nel saturare completamente le risorse a disposizione, andando, ad esempio, ad inserire sempre il maggior numero di pod possibile all'interno dei nodi accesi, e prevedendo un ridimensionamento che vada sempre ad impegnare tutte le risorse per cui sta avvenendo l'esborso di denaro. Risulta, come detto, tuttavia ragionevole dal punto di vista di Azure, in quanto, l'azienda di Microsoft, impegna effettivamente le risorse scelte per il nodo.

Altra caratteristica fondamentale del sistema dei costi delle VM di Azure riguarda l'esistenza di diverse policy per cui questi possono variare. Microsoft permette di scegliere tra tre meccanismi di pagamento differenti:

- *Pay as you go*, meccanismo attraverso il quale è possibile pagare le VM al loro prezzo base (con eventuale costo della licenza in caso di macchine Windows), in base all'effettivo utilizzo, in secondi delle stesse;
- *1 year reserved*, meccanismo attraverso il quale è possibile riservare una VM per un anno, pagandola dunque per l'utilizzo di un anno intero, ma con degli sconti proposti da Azure, nello specifico viene offerta l'eventuale licenza del sistema operativo e viene effettuato uno sconto sul prezzo base della macchina del 40% [15];
- *3 year reserved*, meccanismo attraverso il quale è possibile riservare una VM per tre anni, impegnandosi dunque al pagamento di tre anni interi, ma con il 62% di sconto sul prezzo base della macchina e la concessione gratuita della licenza del sistema operativo [15].

4.2 Configurazioni

Nel capitolo 3, sebbene esso parli dell'architettura della soluzione, non sono state inserite, di proposito, cifre riguardanti le risorse necessarie a ciascun componente. Questa scelta è dettata dal fatto che, la soluzione proposta è estremamente flessibile e, per quanto importanti siano le risorse garantite a ciascun pod, essa trae vero vantaggio dalla parallelizzazione del processo di rendering. È comunque importante sottolineare che, esistono delle condizioni ottimali per ciascun componente in gioco.

Per condizione ottimale, in questo caso, si intende una configurazione di richiesta e limite delle risorse per pod tale per cui una diminuzione porterebbe ad un deterioramento delle performance, mentre un aumento, non garantirebbe un miglioramento dei tempi di elaborazione. È possibile osservare quali siano le configurazioni ottimali di ciascun componente in Tabella 6, Tabella 7 e Tabella 8.

Tabella 6: Richieste e limiti risorse MasterNode

	CPU (millis)	**RAM**
Richiesta	1500	Massimo ottenibile
Limite	2000	Massimo ottenibile

Tabella 7: Richieste e limiti risorse RenderEngine

	CPU (millis)	**RAM**
Richiesta	1200	2500Mi
Limite	1800	6500Mi

Tabella 8: Richieste e limiti risorse FfmpegNode

	CPU (millis)	**RAM**
Richiesta	3000	Massimo ottenibile
Limite	3500	Massimo ottenibile

Alla luce di quanto affermato nelle righe precedenti, è possibile studiare, dunque, assieme al cliente, quale sia il giusto compromesso tra l'esborso di denaro richiesto e le performance che lo stesso vuole garantire, utilizzando il tool *Azure Pricing Calculator*, e tenendo in considerazione che, chiaramente, nodi più potenti sono caratterizzati da costi più elevati, ma garantiscono una maggiore capacità di parallelizzazione, quindi maggiori performance. Tale considerazione permette di valutare l'opportunità di poter fornire diverse *SLA*[37] ai clienti.

[37] SLA, Service Level Agreement, consente di pattuire con il cliente delle garanzie riguardo al livello del servizio offerto

Capitolo Quinto

Capitolo 5

5 Conclusioni

L'ultimo capitolo della presente Tesi di Laurea Magistrale è volto ad effettuare delle analisi conclusive sul sistema ideato ed implementato. Viene dapprima effettuato un confronto tra quello che è il meccanismo utilizzato attualmente da Algo rispetto a quello proposto, segue poi una considerazione su quello che è il sistema di Azure Kubernetes Service e su quelle che sono le pesanti limitazioni che esso porta in dote. Si arriva in fine alle possibili migliorie che possono essere apportate al progetto in esame affinché esso possa essere ulteriormente perfezionato.

5.1 Obiettivi raggiunti

Nell'analisi del sistema costruito, emerge che, a parità di configurazione, le performance sono simili e comparabili all'infrastruttura attualmente in uso. Tuttavia, però, nell'infrastruttura presentata si ottiene:

- Un maggiore controllo riguardo a quello che è lo stato del sistema, grazie all'utilizzo di un sistema di logging capace di mantenere traccia di ogni evento e alla dashboard attraverso la quale è possibile monitorare le condizioni e l'utilizzo delle risorse a disposizione;
- Una migliore gestione della coda dei messaggi, grazie all'utilizzo di un broker;
- Una parallelizzazione ottimale grazie all'utilizzo dei pod;
- Maggiore robustezza del sistema.

A tali obiettivi va aggiunto il fatto che l'infrastruttura risulta essere migliorabile per quanto riguarda le performance, intese come il tempo di elaborazione dei video. Come si evince dal grafico in Figura 5.1, nel corso degli anni, Algo ha costantemente ridotto i tempi di elaborazione dei video grazie al monitoring e al perfezionamento della loro infrastruttura. Mantenendo stabile la durata dei video, si è passati da un tempo di elaborazione medio di circa 300 secondi, ad uno di circa 200 secondi in tre anni.

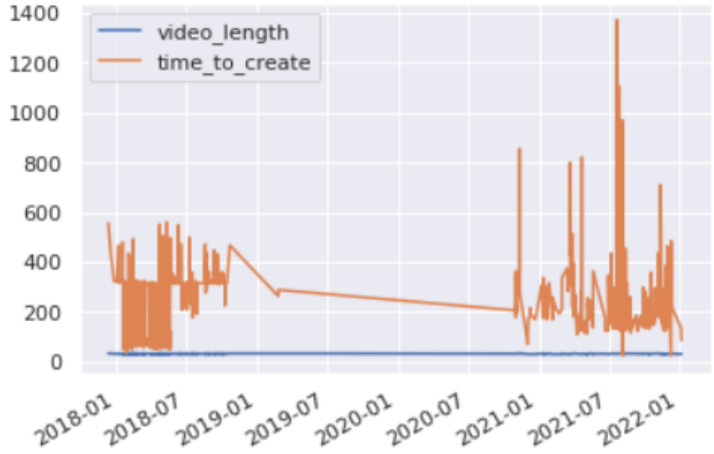


Figura 5.1: Tempo di elaborazione dei video nel corso del tempo

Nel caso in cui poi, si dovesse decidere di andare ad usufruire delle scontistiche proposte da Azure, si riuscirebbe addirittura ad ottenere performance decisamente migliori rispetto a quelle attuali, a prezzi pressocché simili. È inoltre importante sottolineare che, per video di breve durata, attraverso la modalità AVI è possibile ridurre in maniera sostanziale quelle che sono le tempistiche di elaborazione riscontrate dalle analisi dei risultati di Algo.

Si rimane, infine, fiduciosi sul fatto che le evoluzioni tecnologiche, nell'ambito di Azure Kubernetes Service e nell'ambito dell'utilizzo dei container, possano portare ad un miglioramento generale dell'infrastruttura, che non è assicurato dall'utilizzo della meccanica nativa di Adobe Watch Folder. Tale fiducia, la si ottiene dal fatto che il lavoro della community di Kubernetes è incessante ed in continua evoluzione; dunque, di giorno in giorno, vengono annunciate diverse feature che potranno essere sfruttate al fine di ottenere migliori performance.

5.2 Considerazioni

Alla luce di quanto detto nel precedente paragrafo, rimangono comunque dei limiti, superabili solo attraverso quello che è il progresso tecnologico all'interno dell'ambiente Kubernetes e attraverso quello che potrebbe essere un migliore supporto a tale tecnologia.

Ad esempio, rimane evidente che, risulta attualmente impensabile poter usufruire dello scaling dinamico sulla base di un paradigma reattivo, almeno per quanto riguarda i nodi Windows. Questa condizione è dovuta a diversi fattori, tra cui gli attuali tempi di

accensione e, soprattutto, le tempistiche dettate dai pull delle immagini Windows. Tale limitazione può essere abbattuta grazie all'introduzione, tra le immagini preconfezionate all'interno dei nodi Windows di AKS, di immagini compatibili all'installazione di Adobe After Effects.

Altra limitazione dell'attuale supporto alle immagini Windows è relativa all'utilizzo della GPU. Dalla documentazione, pare essere ancora in fase embrionale il supporto dei container basati sul sistema operativo Microsoft a tale risorsa hardware [16]. Azure sta lavorando in questa direzione, e sicuramente, l'introduzione di questo supporto, andrà a ridurre in maniera consistente i tempi di elaborazione medi dei vari video.

È stato poi utilizzato come *cloud provider* Azure, al fine di garantire continuità rispetto a quella che era l'infrastruttura precedente utilizzata da Algo. Rimane tuttavia aperta la possibilità di migrare il sistema su altre piattaforme, al fine di effettuare delle comparazioni, sia dal punto di vista economico, e sia dal punto di vista delle prestazioni. Potrebbero, ad esempio, essere effettuati dei test, e delle analisi dei costi su provider quali *Amazon Web Service* e *Google Cloud Platform*.

5.3 Implementazioni future

Per concludere, si passa ora ad analizzare quelli che sono i possibili perfezionamenti che possono essere apportate al sistema, al fine di renderlo migliore, sia in termini di costi di sviluppo e sia in termini di robustezza e performance.

In primis si può parlare di quello che è il broker che si è deciso di utilizzare. Fin da subito si è puntato a KubeMQ in quanto presentato come broker nativo di Kubernetes. Sebbene esso abbia numerosi punti di forza, analizzati al paragrafo 2.4, presenta la limitazione dovuta al costo. È infatti utile notare che, KubeMQ è utilizzabile attraverso diverse licenze, le quali hanno costi differenti e al contempo garantiscono servizi distinti. La licenza utilizzata all'interno del progetto è quella denominata *community*. Tale licenza viene offerta in maniera gratuita, ma a differenza delle altre, a pagamento, fornisce la possibilità di avere il servizio su un singolo pod, dunque non viene garantita l'alta affidabilità. Per ovviare a questa situazione possono essere intraprese due strade differenti:

- Utilizzare una delle licenze a pagamento fornite da KubeMQ
- Esplorare nuove soluzioni Cloud Native nell'ambito della messagistica, quali ad esempio *NATS* o *Cloudevents*, per citarne alcune, presenti nel CNCF Landscape.

Possono poi essere utilizzati strumenti quali *Helm* e *Terraform* per quanto riguarda la semplificazione dei processi di configurazione, deployment e la gestione automatica dell'intera infrastruttura.

È inoltre possibile attendere il passaggio dei nodi Windows di AKS alla versione 2022, in modo tale da andare, non solo a ridurre in maniera sostanziale quelli che sono i tempi di scaling, in quanto si può facilmente prevedere che l'immagine Docker di Windows 2022 sarebbe tra quelle preconfezionate, ma anche per riuscire a sfruttare in maniera decisamente migliore l'hardware del nodo, e quindi a migliorare ulteriormente le attuali performance.

Infine, attraverso una costante gestione dell'infrastruttura, e attraverso l'analisi dei dati prodotti riguardanti le tempistiche di elaborazione, Sarà possibile trarre delle conclusioni riguardo a quale sia l'effettiva soglia che rende vantaggiosa una modalità di rendering piuttosto che l'altra, e scegliere in maniera automatica quale utilizzare in base, dunque, al contesto in cui si sta lavorando. Tale miglioria può essere però applicata, solo a seguito di un lungo utilizzo dell'infrastruttura e di un bacino di video elaborati molto ampio.

Appendice

Appendice

6 Appendice

6.1 Creazione e configurazione

In questa sezione viene inserita una guida che ha come obiettivo quello di indicare al lettore come si possa, partendo dal solo codice scaricabile da git, realizzare e configurare il cluster analizzato. È opportuno considerare che tutti i passi della presente guida sono applicati a partire da una directory che ha la struttura rappresentata nella Figura 6.1.

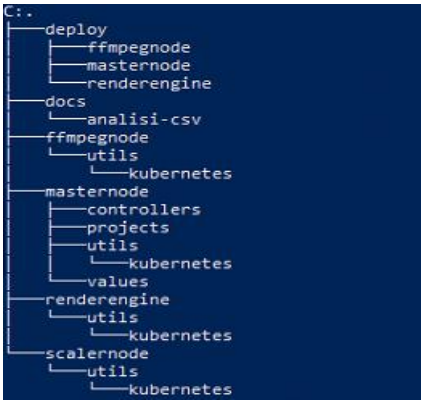


Figura 6.1: Struttura cartella di partenza

6.1.1 Requisiti

La seguente guida prevede l'utilizzo di diversi strumenti, è dunque fondamentale rispettare tutti i requisiti.

- Macchina con sistema operativo Linux Based;
- Macchina con sistema operativo Windows 10 o 11 versione Professional, o Windows Server 2019 o 2022;
- È necessaria l'installazione di "Docker"[38] su entrambi i sistemi operativi;
- È necessaria l'installazione di "Kubectl"[39] su almeno un sistema operativo;
- È necessaria l'installazione di "Azure CLI[40]" su almeno un sistema operativo;
- È necessario disporre di un account Azure con privilegi tali da poter creare e gestire in autonomia servizi relativi ai Container e ai cluster Kubernetes;

[38] https://www.docker.com/products/docker-desktop
[39] https://kubernetes.io/docs/tasks/tools/
[40] https://docs.microsoft.com/it-it/cli/azure/install-azure-cli

6.1.2 Creazione Azure Container Registry

È possibile creare un Container Registry privato sia attraverso linea di comando, per mezzo del tool Azure CLI, e sia attraverso l'interfaccia Web di Azure. Nella presente guida sarà analizzata la seconda metodologia.

1) Recarsi sul portale di Azure[41] ed effettuare login;

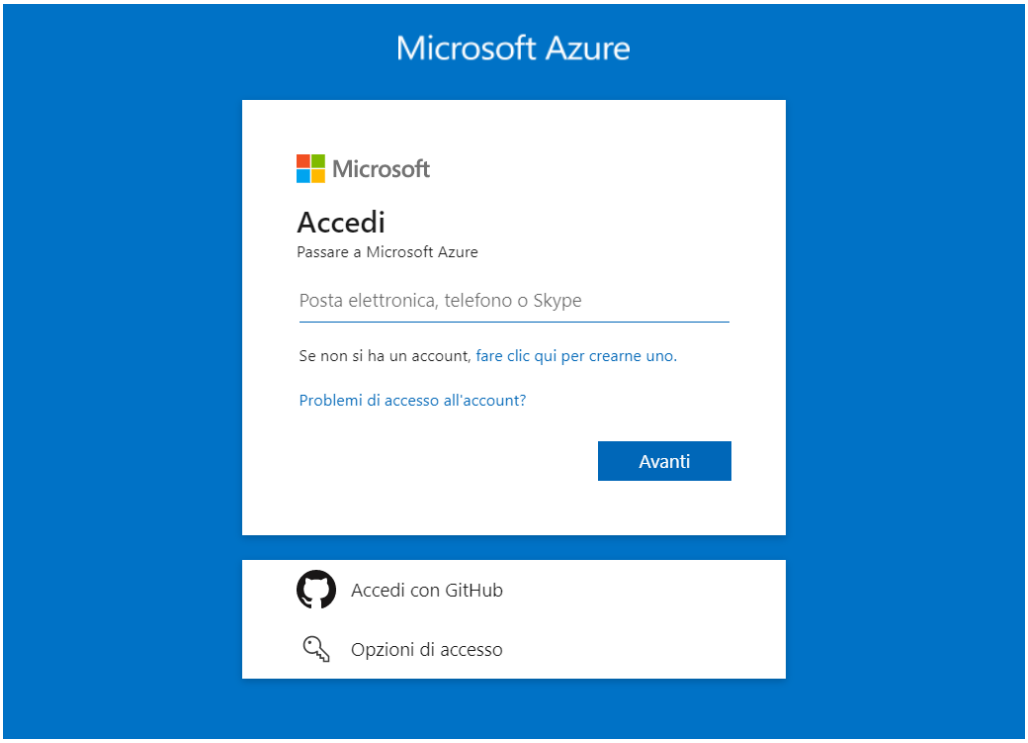


Figura 6.2: Screenshot login Azure

2) In alto, sotto la voce "Servizi di Azure" cliccare sull'icona "Crea una risorsa", come indicato in Figura 6.3;

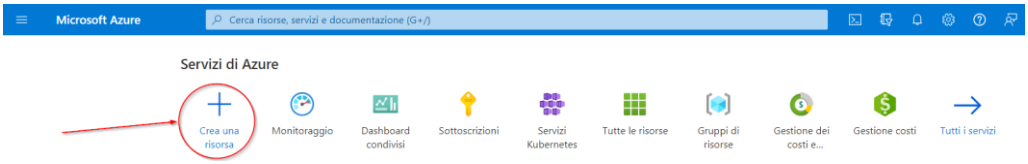


Figura 6.3: Pulsante "Crea"

3) Nella seguente schermata, cercare "Container Registry" e successivamente cliccare su "Crea";

[41] https://portal.azure.com/#home

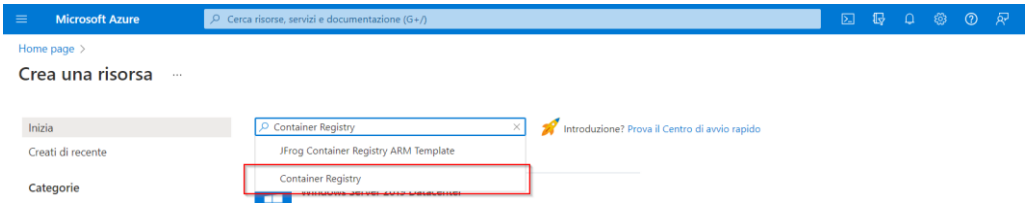


Figura 6.4: Ricerca di "Container Registry"

4) Si avrà dunque accesso ad una pagina nella quale sarà possibile indicare le configurazioni del Container Registry che si vuole creare, Figura 6.5;

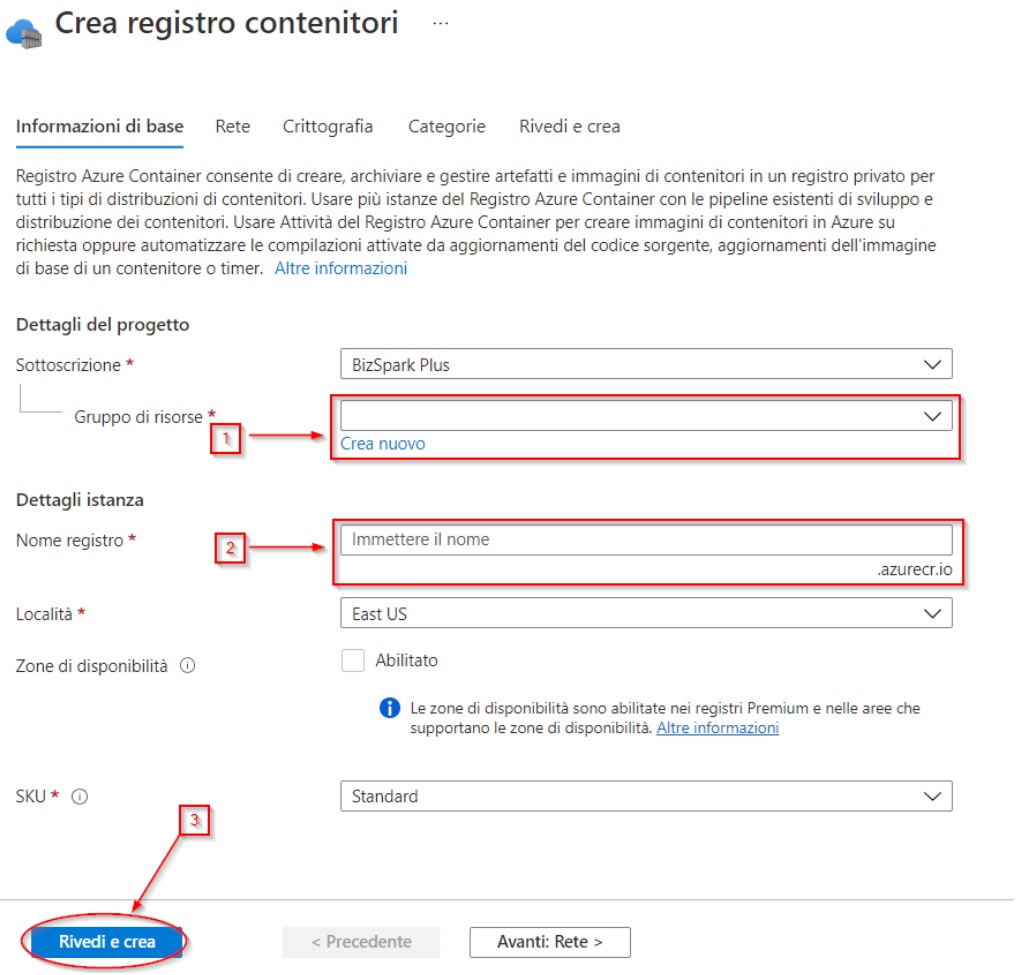


Figura 6.5: Configurazione Azure Container Registry

5) Seguire la procedura guidata indicando il gruppo risorse, il nome del registro e cliccando su “Rivedi e crea”, poi confermare la creazione attraverso il pulsante “Crea”;
6) Per semplicità, nel corso di questa guida, si assumerà che il nome assegnato al Registry sia “acrtest”.

6.1.3 Creazione Docker Image Linux

Partendo dal root della directory la cui struttura è analizzata in Figura 6.1, è possibile creare la Docker image Linux, su sistema operativo Linux Based, attraverso i comandi mostrati in Figura 6.6.

```
1  # Login
2  az login
3  az acr login --name azurecr
4  # Creazione e push docker image masternode
5  cd ./masternode
6  docker build . -t acrtest.azurecr.io/masternode:v1.0.0
7  docker push acrtest.azurecr.io/masternode:v1.0.0
8  cd ..
9  # Creazione e push docker image ffmpegnode
10 cd ./ffmpegnode
11 docker build . -t acrtest.azurecr.io/ffmpegnode:v1.0.0
12 docker push acrtest.azurecr.io/ffmpegnode:v1.0.0
13
```

Figura 6.6: Comandi creazione delle immagini Docker MasterNode e FfmpegNode

Righe 1-2: Viene effettuato dapprima il login attraverso la "Azure CLI" con il comando alla *riga 1*. A seguito di tale comando viene mostrata una pagina web per mezzo della quale è possibile autenticarsi tramite le credenziali Azure. Successivamente, per mezzo del comando alla *riga 2* è possibile accedere alla Azure Container Registry.

Righe 5-8: Ci si reca all'interno della cartella "masternode", si crea e si effettua il push dell'immagine relativa all'omonimo pod all'interno della Azure Container Registry[42].

Righe 10-12: Ci si reca all'interno della cartella "ffmpegnode", si crea e si effettua il push dell'immagine relativa all'omonimo pod all'interno della Azure Container Registry42[42].

6.1.4 Creazione Docker Image Windows

La creazione di un'immagine Windows, necessaria per la realizzazione del solo pod *RenderEngine*, avviene in maniera analoga a quanto esplicato per quanto riguarda la creazione delle immagini docker Linux. Esistono tuttavia degli accorgimenti da rispettare affinché tutto funzioni correttamente:

[42] È fondamentale che il nome dell'immagine inizi con il riferimento alla Registry creata. Nel caso in cui si stia seguendo pedissequamente tale guida, il riferimento sarà "acrtest.azurecr.io", dunque il nome dell'immagine dovrà essere simile a "acrtest.azurecr.io/immagine:v1.0.0";

- È necessario operare all'interno di una macchina che abbia come sistema operativo una delle versioni Windows citate nei requisiti al paragrafo 6.1.1;
- È necessario che, Docker sia impostato in modalità "Windows Container";
    - Per impostare Docker in tale modalità, basta cliccare con il tasto destro sull'icona Docker nella barra delle applicazioni in basso a destra, e cliccare sulla voce "Switch to Windows containers", come mostrato in Figura 6.7;

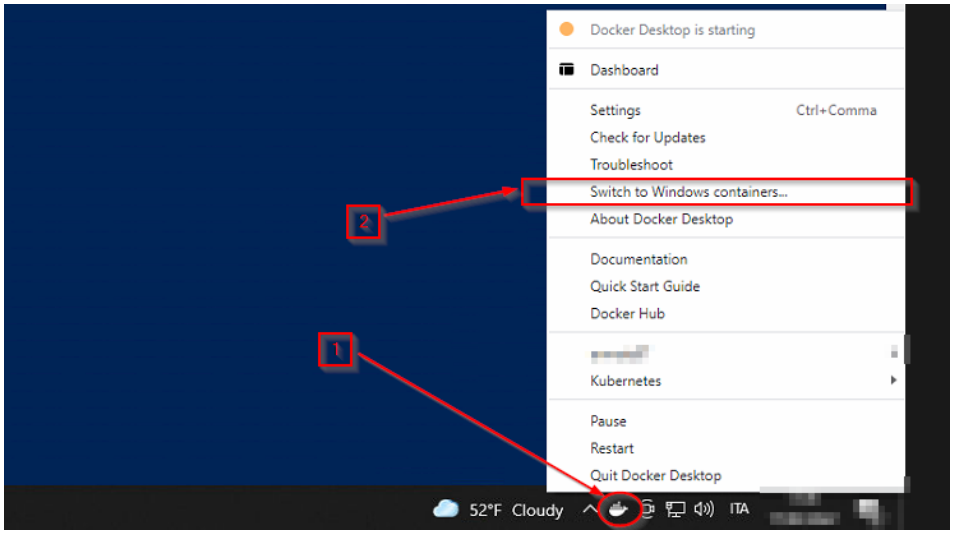


Figura 6.7: Screenshot su come passare alla modalità Windows Containers

- È necessario assicurarsi che tutti i file necessari alla realizzazione dell'immagine siano dentro la cartella dalla quale viene eseguito il comando di build. In Figura 6.9 tale cartella è denominata "BaseRenderEngine" ed essa deve avere la struttura indicata in Figura 6.8. Bisogna dunque assicurarsi che la cartella "/renderengine" della directory in Figura 6.1 venga copiata al suo interno.

```
Adobe After Effects 22.0 Prefs-indep-output.txt
ae_render_only_node.txt
Dockerfile
Adobe
└ ...
renderengine
    aerender.js
    app.js
    CHANGELOG.md
    package.json
    README.md
    utils
        constants.js
        functions.js
        kubernetes
            kubemqClient.js
```

Figura 6.8: Struttura cartella BaseRenderEngine

```
# Login
az login
az acr login --name azurecr
# Creazione e push docker image masternode
cd <path>\BaseRenderEngine
docker build . -t acrtest.azurecr.io/renderengine:v1.0.0
docker push acrtest.azurecr.io/renderengine:v1.0.0

```

Figura 6.9: Comandi creazione RenderEngine

I comandi da eseguire, all'interno di un terminale Powershell, sono dunque indicati in Figura 6.9.

6.1.5 Creazione cluster Kubernetes

Effettuato il push delle immagini all'interno della Azure Container Registry, è possibile ora creare il cluster attraverso il servizio Azure Kubernetes Service. Anche in questo caso è possibile effettuare tale operazione, sia per mezzo del tool "Azure CLI" e sia attraverso l'interfaccia grafica del portale di Azure. All'interno di questa guida verrà analizzata la seconda metodologia.

1) Recatisi all'interno della pagina web di Azure[43] ed effettuato login, in maniera analoga con quanto fatto nella creazione del Container Registry, cliccare su "Crea una risorsa";
2) Cercare poi "Kubernetes Service";
3) Cliccare su "Crea"
4) Seguire la procedura guidata, indicando:
    a. Il gruppo di risorse nel quale si vuole creare il cluster;
    b. Il nome del cluster;
    c. L'area geografica;
    d. La versione di Kubernetes (sarà possibile modificarla anche dopo la fase di creazione);
    e. La dimensione e l'eventuale possibilità di scaling del pool di nodi del control plane;

[43] https://portal.azure.com/#home

f. La dimensione e l'eventuale possibilità di scaling dei pool di nodi worker (sarà possibile aggiungere, rimuovere e modificare i pool di nodi worker anche dopo la fase di creazione del cluster);
g. La configurazione di rete, che dovrà essere "Azure CNI" in quanto "Kubenet" non supporta i container Windows;
h. L'integrazione con il Container Registry creato in precedenza;
i. Eventuali tag;

5) Terminare la procedura di creazione cliccando sul pulsante "Crea";

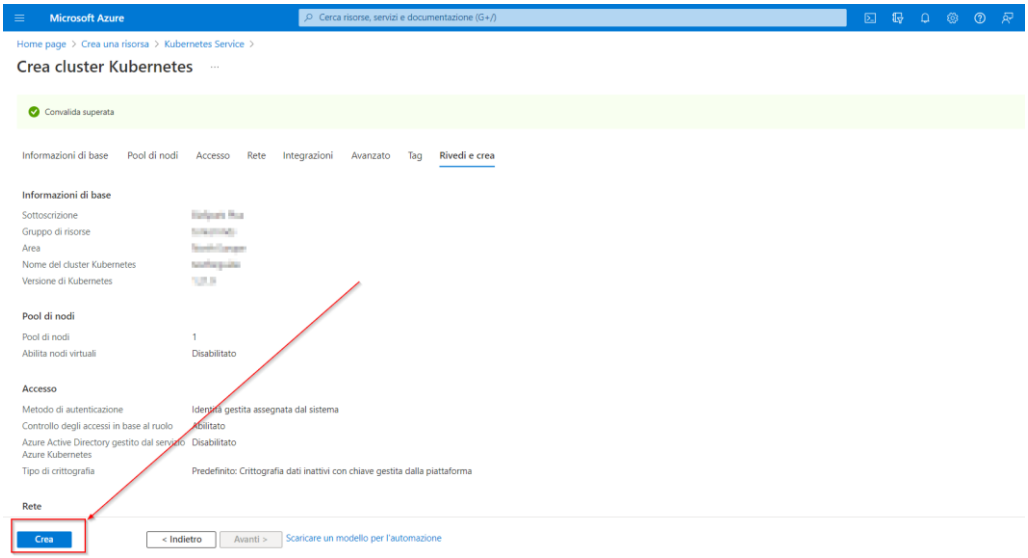


Figura 6.10: Screenshot configurazione cluster Kubernetes

6) Attendere la fine della procedura di creazione della risorsa;
7) Al termine recarsi nella home del cluster e cliccare sulla voce "Connetti", come mostrato in Figura 6.11. Copiare poi i comandi indicati della medesima figura all'interno del terminale ed eseguirli.

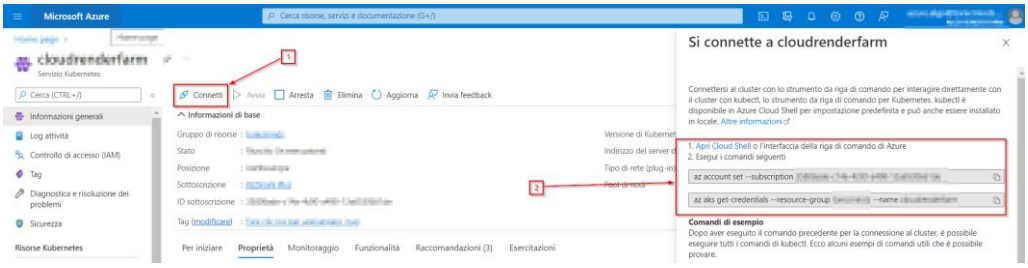


Figura 6.11: Screen home risorse Kubernetes

```
# Login
az login
# Connessione al cluster
az account set --subscription <number_of_subscription>
az aks get-credentials --resource-group <resource_group> --name <name_of_cluster>
# Verifica che tutto abbia funzionato
kubectl config current-context
# Output atteso: <name_of_cluster>

```

Figura 6.12: Elenco comandi creazione cluster

La lista dei comandi, da eseguire tramite terminale, è indicata in Figura 6.12.

6.1.6 Configurazione cluster

A questo punto è possibile configurare il cluster ed eseguire il deploy dell'intera applicazione. I comandi da eseguire in questa fase sono presentati in Figura 6.13, considerando sempre di partire dal root della directory analizzata in Figura 6.1.

```
# Installazione KubeMQ
cd ./deploy
kubectl apply -f kubemq.yaml
# Creazione volumi
kubectl apply -f ./azure-file-sc.yaml
kubectl apply -f ./volumes.yaml
# Creazione ClusterRole
kubectl create clusterrolebinding default-view --clusterrole=view --serviceaccount=default:default
# Creazione infrastruttura
kubectl apply -f ./masternode/deployment.yaml
kubectl apply -f ./masternode/service.yaml
kubectl apply -f ./renderengine/deployment.yaml
kubectl apply -f ./ffmpegnode/deployment.yaml

```

Figura 6.13: Elenco comandi configurazione cluster

Righe 2-3: Il primo step da eseguire, dopo essersi spostati all'interno della cartella "/deploy" consiste nell'installazione di KubeMQ all'interno del cluster.

Righe 5-6: Bisogna poi creare i volumi, i quali utilizzano una StorageClass creata ad hoc. Tale StorageClass viene creata attraverso il manifest "azure-file-sc.yaml", mentre i volumi sono creati attraverso il file yaml "volumes.yaml".

Riga 8: Si passa poi alla creazione di un ClusterRole, affinché i vari pod possano usare le informazioni relative al cluster Kubernetes.

Righe 10-13: È ora possibile creare i servizi e i pod dell'infrastruttura.

A questo punto il sistema è attivo e funzionante, ed è possibile verificare tale situazione attraverso il comando "kubectl get all" che dovrebbe fornire un output simile a quello mostrato in Figura 6.14.

```
[redacted]:~$ kubectl get all
NAME                                          READY   STATUS    RESTARTS   AGE
pod/ffmpegnode-fcb6fc8b6-vv8p9                1/1     Running   0          23h
pod/kubemq-0                                  1/1     Running   0          24h
pod/masternode-deployment-65f848558d-9ppqr    1/1     Running   0          23h
pod/renderengine-deployment-85cf46f9b4-hwzdr  1/1     Running   0          23h
pod/renderengine-deployment-85cf46f9b4-k2zgh  1/1     Running   0          23h
pod/renderengine-deployment-85cf46f9b4-qq28p  1/1     Running   0          23h
pod/renderengine-deployment-85cf46f9b4-smcdl  1/1     Running   0          23h
pod/renderengine-deployment-85cf46f9b4-vh5tn  1/1     Running   0          23h

NAME                        TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)                        AGE
service/kubemq              ClusterIP      10.0.8.227    <none>          50000/TCP,8080/TCP,9090/TCP    24h
service/kubernetes          ClusterIP      10.0.0.1      <none>          443/TCP                        85d
service/masternode-service  LoadBalancer   10.0.125.17   [redacted]      3000:31990/TCP                 59d

NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ffmpegnode               1/1     1            1           23h
deployment.apps/masternode-deployment    1/1     1            1           23h
deployment.apps/renderengine-deployment  5/5     5            5           23h

NAME                                                DESIRED   CURRENT   READY   AGE
replicaset.apps/ffmpegnode-fcb6fc8b6                1         1         1       23h
replicaset.apps/masternode-deployment-65f848558d    1         1         1       23h
replicaset.apps/renderengine-deployment-85cf46f9b4  5         5         5       23h

NAME                      READY   AGE
statefulset.apps/kubemq   1/1     24h
```

Figura 6.14: Output tipo a seguito dell'inserimento del comando "kubectl get all"

6.2 Rendering di un progetto

Considerando che, la meccanica attraverso la quale il progetto[44] After Effects debba essere trasferito all'interno dell'infrastruttura non è oggetto della soluzione richiesta, ma dipende dall'integrazione della stessa all'interno dell'ecosistema per il quale l'infrastruttura deve essere adattata, tale passaggio avviene attraverso il comando "kubectl cp" come mostrato in Figura 6.15.

```
1  # Ricavare il nome di un pod MasterNode
2  kubectl get pods
3  # Il nome del pod scelto sarà indicato nell'esempio come "masternode".
4  # La cartella, nella quale dovrebbe essere presente il progetto per cui effettuare
5  # il rendering, sarà indicata nell'esempio come "project_dir" accessibile tramite "project_path".
6  kubectl cp project_path/project_dir masternode:/usr/src/app/projects/
```

Figura 6.15: Comandi trasferimento progetto all'interno dell'infrastruttura

Al fine di far partire il processo di rendering è a questo punto necessario effettuare una richiesta POST ad uno dei due endpoint esposti dal masternode. Per fare ciò è possibile utilizzare un qualsiasi client capace di effettuare richieste http. Nel presente esempio viene utilizzato il tool Postman[45].

La richiesta deve essere effettuata indicando come indirizzo quello relativo al "masternode-service" nella colonna "EXTERNAL-IP", ottenibile attraverso il comando

[44] N.B. Il nome della cartella contenente il progetto deve avere formato "Project_ID"
[45] https://www.postman.com/

"kubectl get all" come in Figura 6.14 o "kubectl get svc". La porta deve essere impostata a 3000 e il JSON da passare attraverso il body deve essere simile a quello indicato in Figura 6.16. È poi possibile scegliere tra /renderPNG e /renderAVI. La risposta a tale richiesta è simile a quella mostrata in Figura 6.17

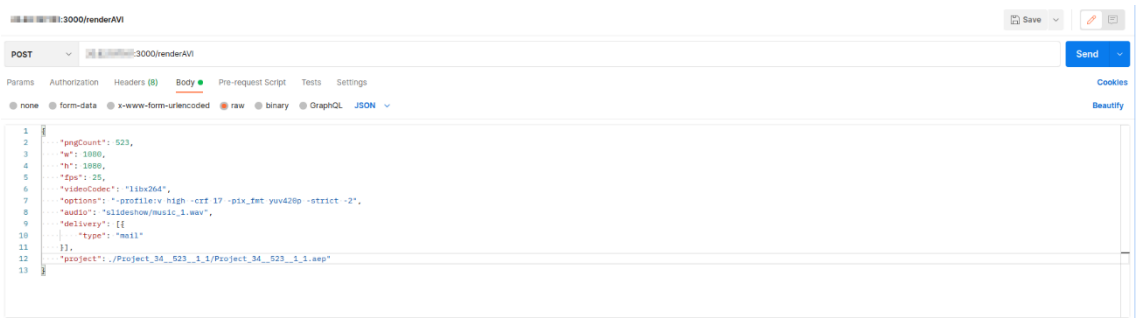


Figura 6.16: Screenshot Postman

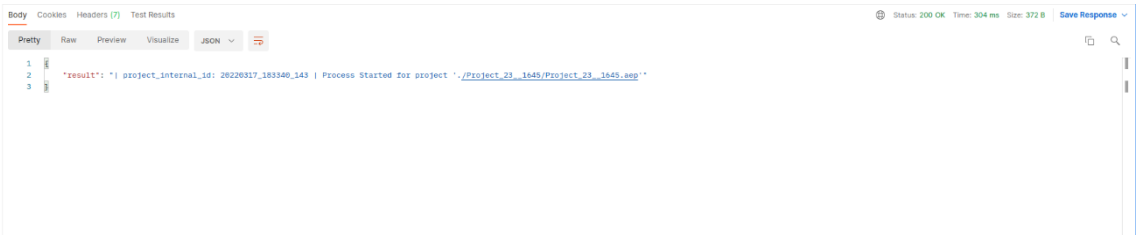


Figura 6.17: Esempio risposta da parte del MasterNode

È possibile monitorare l'andamento del rendering attraverso la dashboard realizzata all'interno del servizio di Azure, la quale mostra i log ma in maniera non ancora completamente ordina, oppure attraverso i comandi indicati in Figura 6.18, Figura 6.19 e Figura 6.20.

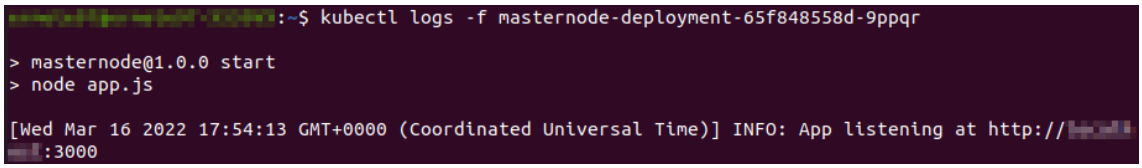


Figura 6.18: Comando per ottenere i log del MasterNode

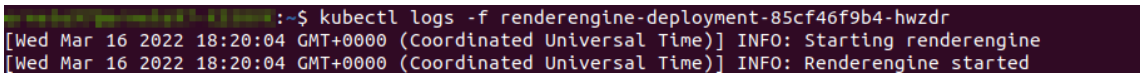


Figura 6.19: Comando per ottenere i log del RenderEngine

```
[redacted]:~$ kubectl logs -f ffmpegnode-fcb6fc8b6-vv8p9

> ffmpegnode@1.0.0 start
> node app.js

[Wed Mar 16 2022 18:20:45 GMT+0000 (Coordinated Universal Time)] INFO: Starting ffmpegnode...
[Wed Mar 16 2022 18:20:45 GMT+0000 (Coordinated Universal Time)] INFO: Ffmpegnode started
```

Figura 6.20: Comando per ottenere i log di FfmpegNode

Di seguito, in Figura 6.21, Figura 6.22 e Figura 6.23, alcuni esempi di log durante l'esecuzione di una operazione di rendering.

```
[Thu Mar 17 2022 18:33:40 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 2022031
7_183340_143 | Received: {"pngCount":1645,"w":1920,"h":1080,"fps":25,"videoCodec":"libx264","options"
:"-profile:v high -crf 17 -pix_fmt yuv420p -strict -2","audio":"slideshow/music_1.wav","delivery":[{"
type":"mail"}],"project":"./projects/Project_23__1645/Project_23__1645.aep","creation_timestamp":1647
542020143,"project_id":5,"internal_id":"20220317_183340_143"}
[Thu Mar 17 2022 18:33:40 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 2022031
7_183340_143 | Process starting for project './projects/Project_23__1645/Project_23__1645.aep'
[Thu Mar 17 2022 18:33:40 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 2022031
7_183340_143 | Output folder for 'Project_23__1645' created
[Thu Mar 17 2022 18:33:40 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 2022031
7_183340_143 | Number of active renderegine in default: 5
[Thu Mar 17 2022 18:33:40 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 2022031
7_183340_143 | Splitting factor for project ./projects/Project_23__1645/Project_23__1645.aep: 5
[Thu Mar 17 2022 18:33:40 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 2022031
7_183340_143 | Sending message to ffmpegnode for project './projects/Project_23__1645/Project_23__164
5.aep'...
[Thu Mar 17 2022 18:33:40 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 2022031
7_183340_143 | Sending messages to render engines for project './projects/Project_23__1645/Project_23
__1645.aep'...
[Thu Mar 17 2022 18:33:40 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 2022031
7_183340_143 | Messages to render engines for project './projects/Project_23__1645/Project_23__1645.a
ep' sent
[Thu Mar 17 2022 18:33:40 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 2022031
7_183340_143 | Message to ffmpegnode for project './projects/Project_23__1645/Project_23__1645.aep' s
ent
```

Figura 6.21: Esempio log MasterNode

```
| project_internal_id: 20220317_183340_143 | aerender.exe: PROGRESS:  0:00:13:03 (328): 0 Seconds

| project_internal_id: 20220317_183340_143 | aerender.exe: PROGRESS:  0:00:13:04 (329): 0 Seconds

| project_internal_id: 20220317_183340_143 | aerender.exe: PROGRESS:  3/17/2022 6:36:16 PM: Finished composition "MAIN".


[Thu Mar 17 2022 18:36:16 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 20220317_183340_143 | Frames 1-329 for project 'Project_23__1645' completed
[Thu Mar 17 2022 18:36:16 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 20220317_183340_143 | Message for project 'Project_23__1645' acked
[Thu Mar 17 2022 18:36:16 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 20220317_183340_143 | File OK1 for project '' created
```

Figura 6.22: Esempio log RenderEngine

```
aep'
| project_internal_id: 20220317_183340_143 | ffmpeg: frame= 1017 fps=4.8 q=22.0 size=   57344kB time=
00:00:38.68 bitrate=12144.8kbits/s speed=0.182x
| project_internal_id: 20220317_183340_143 | ffmpeg: frame= 1018 fps=4.8 q=22.0 size=   57344kB time=
00:00:38.72 bitrate=12132.3kbits/s speed=0.181x
| project_internal_id: 20220317_183340_143 | ffmpeg: frame= 1027 fps=4.8 q=22.0 size=   57344kB time=
00:00:39.08 bitrate=12020.5kbits/s speed=0.182x
| project_internal_id: 20220317_183340_143 | ffmpeg: frame= 1028 fps=4.8 q=22.0 size=   57344kB time=
00:00:39.12 bitrate=12008.2kbits/s speed=0.182x
| project_internal_id: 20220317_183340_143 | ffmpeg: frame= 1037 fps=4.8 q=22.0 size=   57344kB time=
00:00:39.48 bitrate=11898.7kbits/s speed=0.182x
| project_internal_id: 20220317_183340_143 | ffmpeg: frame= 1038 fps=4.8 q=22.0 size=   57344kB time=
00:00:39.52 bitrate=11886.7kbits/s speed=0.182x
[Thu Mar 17 2022 18:38:08 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: 2022031
7_183340_143 | Message visibility extended for project './projects/Project_23__1645/Project_23__1645.
aep'
```

Figura 6.23: Esempio log FfmpegNode

6.3 FAQ

6.3.1 Come scaricare i video elaborati?

L'output risultante dal processo consiste nel video realizzato, in formato mp4, secondo le opzioni scelte attraverso il JSON inviato nella richiesta. Considerando che, analogamente a quanto detto per la fase di trasferimento del progetto all'interno dell'infrastruttura, la gestione del file di output dipende dall'integrazione della soluzione nell'ecosistema per il quale la stessa deve essere adattata, il download del prodotto finale avviene attraverso il comando "kubectl cp", come indicato in Figura 6.24.

```
# Ricavare il nome di un pod FfmpegNode
kubectl get pods
# Il nome del pod scelto sarà indicato nell'esempio come "ffmpegnode".
# Il nome del prodotto finale che si vuole scaricare sarà invece indicato nell'esempio
# come AAAAMMDD_HHMMSS_millis.mp4.
# N.B. Il nome del file di output è equivalente a "<internal_id>.mp4"
kubectl cp ffmpegnode:/usr/src/app/output/AAAAMMDD_HHMMSS_millis.mp4 <path_dst>/<name_dst>.mp4

```

Figura 6.24: Comandi per scaricare video finale

6.3.2 Come accedere alle statistiche?

Altra caratteristica della procedura di rendering di un video, è il salvataggio delle relative statistiche riguardanti i tempi di elaborazione. Tali statistiche possono essere sia scaricate in maniera analoga a quanto avviene per i video e sia accedute direttamente su un pod FfmpegNode attraverso i comandi mostrati in Figura 6.25. Ogni istanza di FfmpegNode crea un suo file di statistiche per evitare race conditions. Ogni file di statistiche è denominato "<*timestamp*>*-statistics.csv*> dove il timestamp corrisponde al momento in cui il file viene creato.

```
# Ricavare il nome di un pod FfmpegNode
kubectl get pods
# Il nome del pod scelto sarà indicato nell'esempio come "ffmpegnode".
kubectl exec --stdin --tty ffmpegnode -- /bin/bash
# A questo punto i comandi vengono inseriti direttamente all'interno del pod
# Tale situazione è rappresentata in questo esempio attraverso il prefisso "ffmpegnode#>"
ffmpegnode#> cat /stats/<timestamp>-statistics.csv

```

Figura 6.25: Comandi per accedere alle statistiche

6.3.3 Come interpretare i log?

I log, a prescindere dal pod che li genera ha sempre la medesima struttura. Tale struttura, osservabile in Figura 6.26, comprende nell'ordine:

- Informazioni temporali;
- Livello del log[46];
- Id interno del progetto[47];
- Messaggio.

```
# Esempio di log
[Thu Mar 17 2022 18:33:40 GMT+0000 (Coordinated Universal Time)] INFO: | project_internal_id: <id> | message

```

Figura 6.26: Esempio tipico di log

6.3.4 Come cancellare file?

Per cancellare i file di statistiche, progetti o video, basta collegarsi ad un pod connesso al volume contenente i file che si vogliono eliminare ed effettuare una "remove". La Figura 6.27 mostra come eliminare un file di statistiche.

```
# Ricavare il nome di un pod FfmpegNode
kubectl get pods
# Il nome del pod scelto sarà indicato nell'esempio come "ffmpegnode"
kubectl exec --stdin --tty ffmpegnode -- /bin/bash
# A questo punto i comandi vengono inseriti direttamente all'interno del pod
# Tale situazione è rappresentata in questo esempio attraverso il prefisso "ffmpegnode#>"
ffmpegnode#> rm /stats/<timestamp>-statistics.csv

```

Figura 6.27: Esempio eliminazione file statistiche

6.3.5 Come recuperare inconsistenze?

Considerando che l'infrastruttura realizzata è ancora in versione prototipale, è possibile in alcuni casi che si verifichino situazioni di inconsistenza. In questi casi può venire in aiuto l'endpoint di MasterNode "/ackall" che si occupa di svuotare la coda dei messaggi.

[46] Il livello del log può essere: INFO, WARNING, ERROR.
[47] L'informazione relativa all'id interno è presente solo se il log si riferisce all'elaborazione di un progetto.

Bibliografia

[1] Wikipedia, «Cloud Computing - Wikipedia,» Wikipedia, 13 Febbraio 2020. [Online]. Available: https://it.wikipedia.org/wiki/Cloud_computing. [Consultato il giorno 20 Febbraio 2022].

[2] RedHat, «Le principali differenze tra IaaS, PaaS e SaaS,» RedHat, 2 Aprile 2020. [Online]. Available: https://www.redhat.com/it/topics/cloud-computing/iaas-vs-paas-vs-saas. [Consultato il giorno 20 Febbraio 2022].

[3] A. Salgarelli, «Cos'è la CNCF o Cloud Native Computing Foundation e qual è il suo ruolo?,» Kiratech, 1 Luglio 2019. [Online]. Available: https://www.kiratech.it/blog/cosa-la-cncf-o-cloud-native-computing-foundation-e-quale-il-suo-ruolo. [Consultato il giorno 21 Febbraio 2022].

[4] Kubernetes, «What is Kubernetes?,» Kubernetes, 23 Giugno 2021. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Consultato il giorno 22 Febbraio 2022].

[5] Docker, «What is a Container | App Containerization | Docker,» Docker, [Online]. Available: https://www.docker.com/resources/what-container. [Consultato il giorno 21 Febbraio 2022].

[6] Kubernetes, «Kubernetes Components | Kubernetes,» Kubernetes, 21 Febbraio 2022. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Consultato il giorno 22 Febbraio 2022].

[7] Kubernetes, «Pods | Kubernetes,» Kubernetes, 17 Ottobre 2021. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/. [Consultato il giorno 22 Febbraio 2022].

[8] K. Bobrov, Asynchronous programming, 2020.

[9] KubeMQ, «Introduction - KubeMQ Docs,» KubeMQ, 2021. [Online]. Available: https://docs.kubemq.io/. [Consultato il giorno 23 Febbraio 2022].

[10] KubeMQ, «Kafka vs KubeMQ | Which is best for Microservices and Kubernetes,» KubeMQ, [Online]. Available: https://kubemq.io/kafka-vs-kubemq/. [Consultato il giorno 23 Febbraio 2022].

[11] M. Cracan, «Web REST API Benchmark on a Real Life Application | by Mihai Cracan | Medium,» Medium, 2 Agosto 2017. [Online]. Available: https://medium.com/@mihaigeorge.c/web-rest-api-benchmark-on-a-real-life-application-ebb743a5d7a3. [Consultato il giorno 24 Febbraio 2022].

[12] KubeVirt, «Containerized Data Importer - KubeVirt User-Guide,» KubeVirt, [Online]. Available: https://kubevirt.io/user-guide/operations/containerized_data_importer/. [Consultato il giorno 25 Febbraio 2022].

[13] «Windows by Microsoft | Docker Hub,» Docker Hub, [Online]. Available: https://hub.docker.com/_/microsoft-windows. [Consultato il giorno 1 Marzo 2022].

[14] WineHQ, «WineHQ - Run Windows applications on Linux, BSD, Solaris and macOS,» WineHQ, [Online]. Available: https://www.winehq.org/. [Consultato il giorno 1 Marzo 2022].

[15] Microsoft, «Azure Reserved Virual Machine Instances | Microsoft Azure,» Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/pricing/reserved-vm-instances/. [Consultato il giorno 3 Marzo 2022].

[16] Microsoft, «Usare GPU nel servizio Azure Kubernetes - Azure Kubernetes Service | Microsoft,» Microsoft, [Online]. Available: https://docs.microsoft.com/it-it/azure/aks/gpu-cluster. [Consultato il giorno 7 Marzo 2022].

[17] I. J. Ben Stopford, «Kafka Without ZooKeeper: A Sneak Peek At the Simplest Kafka Yet,» Confluent, 30 Marzo 2021. [Online]. Available:

https://www.confluent.io/blog/kafka-without-zookeeper-a-sneak-peek/.
[Consultato il giorno 23 Febbraio 2022].