PROGETTO

TESSUTO DIGITALE METROPOLITANO

POR FESR 2014-2020, Azione 1.2.2

Delibera 6614 del 13.12.2016, Progetto Complesso area ICT della S3

Deliverable D3.3 – SISTEMA PER L'AGGREGAZIONE E TRATTAMENTO BIG-DATA E DIFFUSIONE OPEN DATA (VERSIONE PRELIMINARE)	
Data di consegna prevista: 6/2019	**Data di consegna prevista:** 6/2019
Natura: Rapporto e codice	**Versione:** 1.0
Livello di Disseminazione: (PU)	
Sommario	
TDM è un progetto collaborativo tra CRS4 e Università di Cagliari che mira ad offrire nuove soluzioni intelligenti per aumentare l'attrattività cittadina, la gestione delle risorse e la sicurezza e qualità di vita dei cittadini, attraverso lo studio e sviluppo di tecnologie abilitanti e di soluzioni verticali innovative per la protezione dei rischi ambientali, l'efficienza energetica e la fruizione dei beni culturali. In questo deliverable presentiamo la descrizione dettagliata del blocco di processamento ed analisi dati come anche delle scelte tecnologiche prese per realizzarlo.	

REGIONE AUTÒNOMA DE SARDIGNA
REGIONE AUTONOMA DELLA SARDEGNA

POR FESR SARDEGNA 2014-2020

SARDEGNA RICERCHE

UNIONE EUROPEA
Fondo europeo di sviluppo regionale

REPUBBLICA ITALIANA

Redazione			
	Nome	Partner	Data
Autore	Muriel Cabianca, Massimo Gaggero, Enrico Gobbetti, Simone Leo, Marino Marrocu, Marco Enrico Piras, Luca Pireddu, Carlo Podda, Gabriella Pusceddu, Gianluigi Zanetti	CRS4	19/06/2019
Approvato da	G. Zanetti/E. Gobbetti	CRS4	24/06/2019

Storia e contributi			
Vers.	Data	Commento	Autori
V0.1	28/05/2019	Primo draft con organizzazione contenuti	G. Zanetti E. Gobbetti(CRS4)
V1	24/06/2019	Versione finale	G. Zanetti E. Gobbetti(CRS4)

Indice

1 Introduzione

TDM è un progetto collaborativo tra CRS4 e Università di Cagliari che mira ad offrire nuove soluzioni intelligenti per aumentare l'attrattività cittadina, la gestione delle risorse e la sicurezza e qualità di vita dei cittadini, attraverso lo studio e sviluppo di tecnologie abilitanti e di soluzioni verticali innovative per la protezione dei rischi ambientali, l'efficienza energetica e la fruizione dei beni culturali.

Uno degli obiettivi principali del progetto TDM è di realizzare un'architettura scalabile per l'acquisizione, l'integrazione e l'analisi di dati provenienti da sorgenti eterogenee in grado di gestire i dati generati da un'area metropolitana estesa. L'implementazione prevista nel quadro del progetto riguarda casi di studio nell'area metropolitana della città di Cagliari, ma l'obiettivo è quello di generare soluzioni scalabili generali che possano servire da best practice per future implementazioni di servizi in aree geografiche ampie e/o densamente popolate. Il progetto deve quindi combinare, sviluppare ed estendere soluzioni tecnologiche in vari campi, dalla sensoristica ai big data e dalla simulazione alla visualizzazione.

1.1 Architecture Complessiva

L'architettura generale di TDM è strutturata su tre grandi blocchi, dedicati, rispettivamente, all'acquisizione distribuita dei dati, al loro processamento e, infine, alla loro presentazione e distribuzione verso l'esterno.

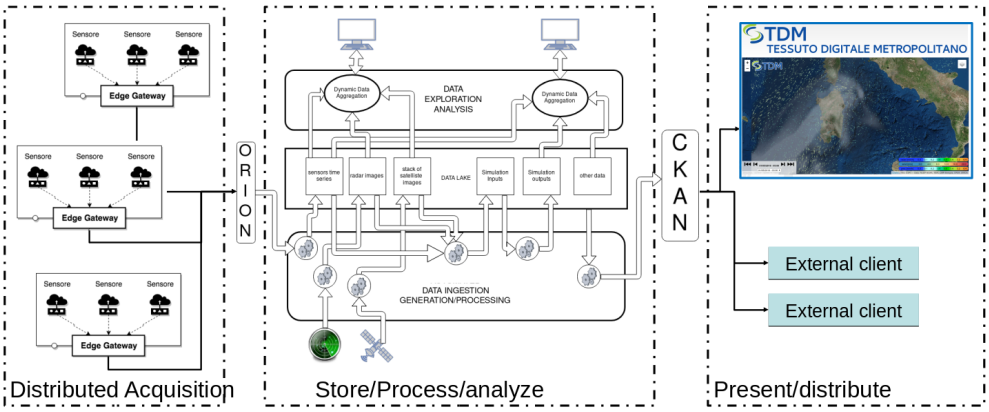

Figura 1.1: TDM general architecture

Il primo blocco, quello a sinistra nella figura, si preoccupa, sostanzialmente, di fornire gli strumenti necessari (edge device, standardizzazione su protocolli e formati dati FIWARE) per

poter agganciare diverse tipologie di sensori (principalmente legati all'acquisizione di informazioni ambientali e consumi elettrici) e convogliarne i risultati in una forma standardizzata verso un context broker, ORION nel nostro caso specifico, che funge da punto di snodo per i flussi di dati acquisiti. Questo blocco è stato descritto in dettaglio dal punto di vista tecnico nel Deliverable D3.1.

Il terzo blocco, quello a destra nella figura, è dedicato alla distribuzione e presentazione dei dati accumulati e generati dal progetto TDM e resi disponibili come dati aperti. Esso è realizzato attraverso una serie di componenti integrati tra di loro, ognuno con caratteristiche e compiti dedicati. Tra di essi i principali sono il portale CKAN, attraverso cui si forniscono, principalmente, informazioni di indicizzazione sui dati disponibili e server di dati veri e propri per cui è previsto un accesso secondo un protocollo di tipo REST. Di seguito descriviamo CKAN e l'interfaccia REST prevista, La sezione 4 presenta dettagli sulle tipologie di dati trattati, e le seguenti esempi su come utilizzare le interfacce e i dati. Questo blocco è stato descritto in dettaglio dal punto di vista tecnico nel Deliverable D3.2.

Il blocco centrale accumula, processa e analizza dati provenienti da varie sorgenti che, oltre a quelli provenienti da sensori distribuiti includono i risultati di simulazioni meteo, immagini satellitari, immagini dal radar meteorologico, mappe e altri dati statici georeferenziati.

1.2 Ambito di questo deliverable

In questo deliverable presentiamo la descrizione dettagliata del blocco di processamento ed analisi dati come anche delle scelte tecnologiche prese per realizzarlo.

1.3 Struttura del report

Il report è organizzato come segue:

- al Capitolo 2 descriviamo l'architettura del sottosistema big-data in termini di funzionalità e componenti;
- al Capitolo 3 presentiamo l'infrastruttura di calcolo sulla quale funziona il sistema, differenziando tra cluster OpenStack e HPC;
- al Capitolo 4 descriviamo il nostro approccio di infrastructure-as-a-code, partendo dal livello di terraforming e descrivendo poi i vari containers e l'implementazione Kubernetes;
- al Capitolo 5 dettagliamo i principali componenti del sottosistema big data di TDM, con particolare riferimento al Data Lake ed all'accesso alla computazione;
- al Capitolo 6 dettagliamo esempi di principali workflow attualmente implementati, ed in particolare la gestione dei flussi di dati ed il running di simulazioni meteo;
- al Capitolo 7 presentiamo risultati di performance del sistema sia in termini di accesso ai dati che di esecuzione di job paralleli a grande scala.

Il rapporto si conclude con un riassunto dei principali obiettivi raggiunti (Capitolo 7).

2 Architettura

2.1 Punto di vista funzionale

La figura 2.1 illustra l'architettura di TDM da punto di vista funzionale. La prima fase è quella dell'ingestione dei dati, in cui si collezionano i dati provenienti dai sensori, dai radar e dai satelliti. Mentre i primi arrivano in modalità push, i restanti vengono recuperati periodicamente in modalità pull.

Una volta ottenuti i dati, questi sono inseriti nello storage (Data Lake), in grado di gestire in maniera scalabile dati eterogenei. Una volta immagazzinati i dati, è possibile eseguire su di essi in maniera automatica aggregazioni, simulazioni e computazione complessa.

I dati e i loro derivati sono quindi analizzabili da esperti e data scientist, che, in maniera interattiva, possono visualizzare, eseguire query e ulteriori computazioni.

Una vista semplificata e immediatamente fruibile dei dati è periodicamente pubblicata sul portale degli open data.

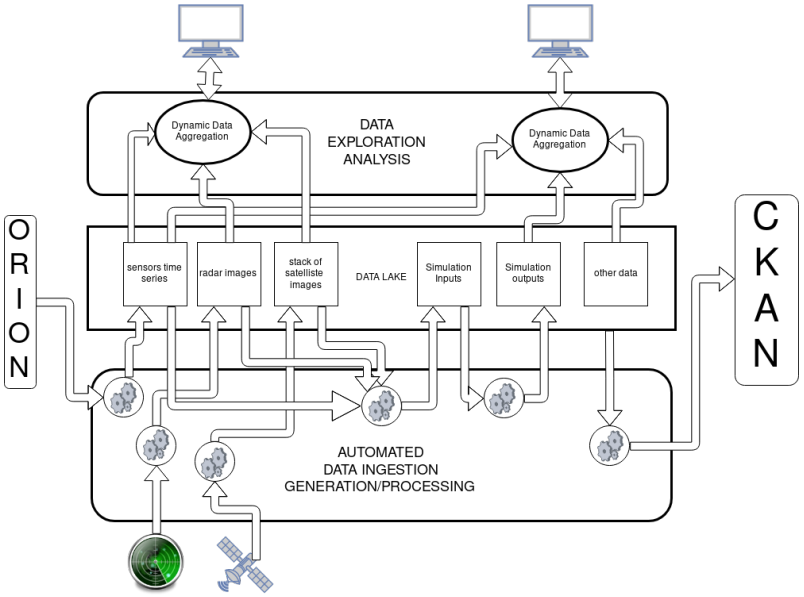

Figura 2.1: Architettura piattaforma TDM; punto di vista funzionale.

2.2 Visione a componenti

L'ingresso dei dati esterni verso l'infrastruttura di calcolo TDM avviene attraverso modalità differenti a seconda del tipo di dato. Per i dati originati da sensori distribuiti quali stazioni meteo, qualità dell'aria o energetiche, la raccolta avviene attraverso componenti e metodiche proprie dell'Internet of Things messe a disposizione dalla piattaforma FIWARE. I messaggi inviati dagli *Edge Gateway* distribuiti sul territorio urbano, i quali a loro volta ricevono e convertono in un modello FIWARE comune i dati delle stazioni di misura, raggiungono la piattaforma di calcolo TDM attraverso il protocollo MQTT, uno tra i maggiormente usati nel IoT, utilizzando un canale autenticato e crittografato con certificati SSL/TLS. Il componente software che effettivamente riceve i messaggi è il *broker MQTT Mosquitto*, il quale funziona anche da disaccoppiamento tra rete esterna e rete interna e permettendo un primo punto di flessibilità. Il protocollo MQTT, così come il broker, sono agnostici rispetto al contenuto e al formato del messaggio. La gestione quindi della rappresentazione dei dispositivi e dei tipi di messaggi è affidata al componente FIWARE *IoTAgent*. Questo è un componente specializzato nel trattare dati proveniente da dispositivi IoT e tra i vari compiti a cui assolve ci sono l'aggiunta dinamica di nuovi dispositivi nel Context Broker e la validazione e l'aggiornamento degli attributi del messaggio. Di fatto è l'interfaccia tra i dispositivi e il componente successivo, il *Context Broker Orion*, cuore della piattaforma FIWARE. Il Context Broker è il componente a cui il resto dell'ecosistema FIWARE fa riferimento per avere informazioni riguardo ai dispositivi afferenti al sistema. Al Context Broker ad esempio possono essere richieste le liste dei dispositivi registrati, il loro tipo e il loro stato più recente, ossia una istantanea dei loro parametri in seguito all'ultimo aggiornamento di questi. Orion inoltre permette di notificare ad altri componenti quando un particolare evento occorre ad un dispositivo o ad un parametro attraverso le notifiche. Un componente può ad esempio sottoscrivere su Orion per una notifica ogni volta che un parametro cambia o quando il suo valore viene aggiornato entro od oltre un determinato range. Di per se il Context Broker non memorizza i valori ricevuti se non l'ultimo aggiornamento. Per avere uno storage persistente è necessario un *Persistence Connector*, ossia un componente che riceve dal Context Broker le notifiche ogni volta che un dispositivo aggiorna i propri parametri e li salva su database o su filesystem. Per tale compito viene usato il Persistence Connector FIWARE *Cygnus*, specializzato per il salvataggio dei dati in arrivo

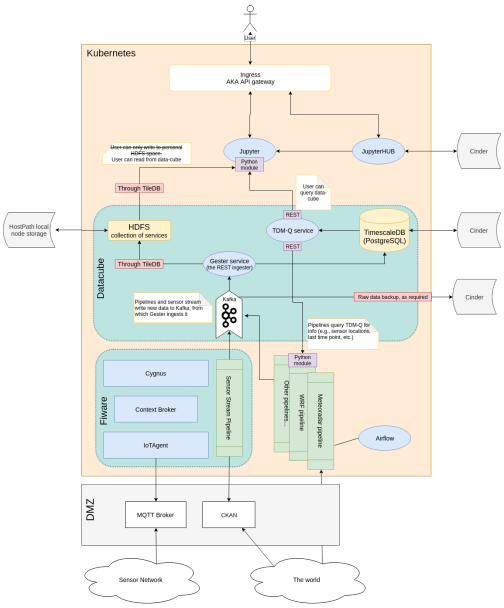

Figura 2.2: Architettura piattaforma TDM: componenti.

sul sistema di code real time e scalabile *Kafka*[1], per la successiva elaborazione. Nel dettaglio, i messaggi NGSI vengono raggruppati in code chiamate topic, una per FIWARE service path. Qui vengono consumati da una serie di job *Flink*[2]. Apache Flink è un framework di ultima generazione per processing real time e distribuito di grosse mole di dati. I job si occupano di eseguire delle aggregazioni temporali (giornaliere e orarie) e memorizzare i dati grezzi e aggregati sul Data Lake e su CKAN. CKAN (Comprehensive Knowledge Archive Network) è uno strumento open source appartenente all'ecosistema Fiware per la creazione di siti web in grado di gestire open data. È analogo a WordPress, ma per i dati. È utilizzato da governi nazionali e locali, istituti di ricerca e in generale organizzazioni che gestiscono grosse moli di dati. Consente la ricerca e il browsing dei dati, nonché la loro preview tramite mappe, grafici e tabelle.

L'elaborazione batch dei dati, sia da sensore che da satellite o radar meteorologico, viene gestita da un componente specifico, il *Workflow Manager* che si occupa dello scheduling, trigger e dell'esecuzione della pipeline. L'implementazione scelta per il Workflow Manager è il software *Apache Airflow* che permette l'esecuzione dei task inerenti alle pipeline sulla stessa infrastruttura Kubernetes della piattaforma di computazione.

La grande mole di dati trattati richiede l'utilizzo di un file system scalabile, robusto ed efficiente. Il componente adottato da TDM per soddisfare queste esigenze è HDFS, un file system distribuito altamente scalabile e particolarmente efficiente nella lettura in streaming di grossi data set. Tale caratteristica si adatta bene a data set che, una volta scritti, non vengono più modificati, ma solo integrati e interrogati, come accade nel caso dei dati TDM.

In TDM, il cluster HDFS viene instanziato automaticamente su Kubernetes per mezzo di un Helm chart, a partire da immagini Docker di Hadoop realizzate dal CRS4. Nell'architettura TDM, il ruolo principale di HDFS è la memorizzazione di grosse moli di dati multidimensionali come, ad esempio, flussi di immagini radar e satellitari. La manipolazione di tali dati su HDFS è mediata da TileDB, una libreria specializzata nella gestione degli array multidimensionali, per assicurare l'efficienza nell'estrazione di opportuni sottoinsiemi di dati (selezioni lungo dimensioni spaziali e/o temporali). TileDB inoltre dispone di un'interfaccia Python, che lo rende agevolmente utilizzabile in uno script o in un notebook Jupyter.

La maggior parte dei dati trattati nell'ambito del progetto sono riferiti a opportuni istanti di tempo e posizioni geografiche, non necessariamente puntiformi. L'interrogazione dei dati deve quindi consentire la selezione di sottoinsiemi lungo le dimensioni spaziali e temporali. A ciò si aggiunge la selezione delle sorgenti in base alle loro proprietà, ad esempio il modello o le grandezze misurata. Nell'ambito di TDM, il componente che svolge questa funzione è TDM-Q.

L'interfaccia di query è realizzata mediante un'API REST. Le principali entità interrogabili sono i sensori, le tipologie di sensori (ad es. di temperatura, di umidità) e le misure (dati rilevati dal sensore). Le query si eseguono tramite opportune chiamate HTTP al servizio web, mentre i risultati sono restituiti in formato JSON. Oltre che dall'esterno, l'API REST può essere utilizzata da parte di altri componenti del sistema, in particolare JupyterHub.

[1] https://kafka.apache.org/
[2] https://flink.apache.org/

Dal punto di vista architetturale, TDM-Q si divide in due componenti principali: il database server, che memorizza dati e metadati, e il front-end, che ospita l'API REST. Il primo è basato su di un database PostgreSQL al quale sono state aggiunte le estensioni TimescaleDB per le serie temporali e PostGIS per gli oggetti geolocalizzati. Il secondo è realizzato mediante un'applicazione Flask servita da Gunicorn. Anche in questo caso, il deployment su Kubernetes è automatizzato tramite Helm e Docker.

3 Infrastruttura di calcolo

Le principali infrastrutture di calcolo utilizzate dal progetto TDM sono il cluster OpenStack ed il cluster HPC gestiti dal Joint Innovation Center.

3.1 Cluster OpenStack JIC

Il JIC fornisce servizi di tipo IaaS tramite la piattaforma opensouce OpenStack[1]. In questo momento, il cluster OpenStack conta 8 nodi di calcolo e 3 nodi di controllo. I nodi di calcolo hanno, ciascuno, 28 core su processori Intel(R) Xeon(R) CPU E5-2683 v3 a 2.00GHz, con 512GB RAM e circa 20TB di disco. Tutti i nodi sono connessi tra loro sia tramite una connessione Ethernet a 10Gbps, che tramite la rete Infiniband a 56Gbps. Le comunicazioni di controllo avvengono su una rete privata mentre i servizi pubblici sono esposti verso la rete JIC tramite due proxy/bilanciatori. Il cluster OpenStack eroga servizi di calcolo tramite istanze virtuali (Nova), di reti virtuali (Neutron), un catalogo di immagini per VM (Glance), spazio disco a blocchi (Cinder) ed un sistema di autenticazione e autorizzazione. In questo momento, sono allocati per TDM l'equivalente di tre nodi fisici suddivisi tra circa venti macchine virtuali. Il vantaggio di OpenStack è che, attraverso di esso, è possibile fornire una interfaccia unificata che astrae l'hardware sottostante e lo rende fruibile attraverso l'utilizzo di un'API. Si raggiunge così un'estrema flessibilità sia lato amministratore che lato utente, che dispone di un'autonomia difficilmente implementabile con soluzioni di tipo Grid.

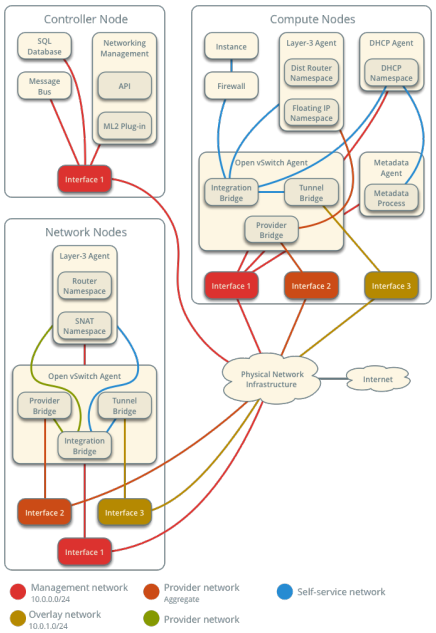

Figura 3.1: Architettura OpenStack e relativa infrastruttura di rete.

Nella figura 3.1 viene illustrata l'architettura generale del sistema OpenStack utilizzato, e la relativa infrastruttura di rete.

[1] https://www.openstack.org/

3.2 Cluster HPC JIC

Il JIC fornisce servizi *classici* di calcolo HPC attraverso il suo cluster HPC. Esso può essere considerato come composto da due parti separate da una divisione di tipo logico che rende possibile all'occorrenza l'ampliamento o estensione dell'una sull'altra e, all'occorrenza, la loro sovrapposizione o unione.

La prima parte, composta da un maggior numero di macchine, è utilizzata per l'HPC classico – tipicamente jobs MPI utilizzati per simulazioni in meteorologia, fluidodinamica e idrodinamica – ed è gestita tramite un sistema di code o Distributed Resources Manager (DRM) che nel nostro caso è Sun Grid Engine (SGE). I nodi di calcolo che ne fanno parte sono caratterizzati da CPU multicore veloci, con un rapporto di RAM per core pari a circa 6GB/core, hard disk locali piccoli e veloci, connessione di rete per dati e servizi a 10GB e connessione Infiniband FDR a 56 Gbps per le esigenze dei programmi paralleli che richiedono una bassa latenza nelle comunicazioni tra nodi. La restante parte del cluster, 16 nodi di tipo Fat Node

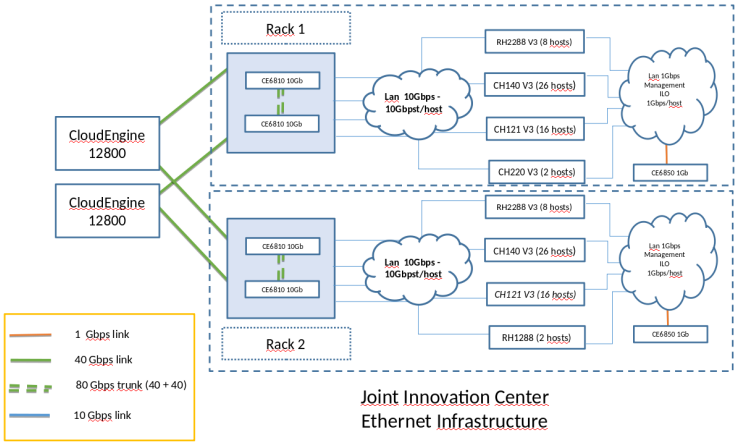

Figura 3.2: Architettura cluster HPC JIC

anch'essi all'occorrenza utilizzabili con SGE, hanno sostanzialmente le stesse caratteristiche di CPU e di rete dei nodi per l'HPC classico, ma hanno molta piú RAM, circa 20GB/core e molti hard disk capienti, 36TB/nodo di spazio raw per nodo. Di queste macchine, attualmente 8 sono gestite con OpenStack (vedi sezione precedente), le restanti macchine come nodi computazionali HPC o come WS di analisi e elaborazione per la pre/post produzione. In totale il cluster JIC dispone di 2560 cores, per un totale di più di 19TB di RAM e 600TB di disco Tutti i nodi sono collegati tra loro tramite una rete ethernet a 10Gbps e una rete Infiniband FDR a 56Gbps. La rete ethernet è totalmente ridondata grazie ad 6 switch eth 10Gbps. La rete Infiniband è attualmente fornita dagli 8 switch integrati negli enclosure che ospitano i nodi. Tutti gli apparati, nodi di calcolo, apparati di rete e i rack refrigerati che ospitano le macchine, hanno almeno due linee di alimentazione che sono collegate a linee elettriche indipendenti.

Sopra SGE, sono stati implentati meccanismi per la creazione di pool dinamici di macchine,

orchestrate tramite il software Kubernetes. Il sistema permette di mettere sotto il controllo di nodi Kubernetes esterni al cluster gruppi arbitrari dei nodi di calcolo disponibili, vedi sezione 5.2.3. Il punto di forza di questa soluzione è che le istanze e le richieste di Kubernetes, con tutto quanto gira al loro interno, sono in grado di lavorare fianco a fianco con tutti gli altri lavori gestiti da SGE, condividendo l'utilizzo delle risorse in maniera dinamica e mediata dal sistema DRM.

4 Infrastructure as Code

TDM segue, dove possibile, la metodologia di *Infrastructure as code* per la configurazione delle risorse di calcolo e dei sistemi in funzione su di essi. L'obiettivo è di disaccoppiare il più possibile la definizione delle configurazioni richieste dalle specifiche dei sistemi – ad esempio OpenStack, AWS, Azure – su cui esse dovranno essere materializzate.

Infrastructure as Code indica un approccio programmatico alla gestione e al provisioning di infrastrutture per la computazione, basato su file di scripting e di configurazione machine readable, in opposizione ad una configurazione hardware ad hoc e a strumenti interattivi che tipicamente conducono all'effetto *snowflake*, per cui ogni ambiente diventa unico e irripetibile. Tramite un modello descrittivo dell'infrastruttura che sia mantenibile e versionabile esattamente come il codice sorgente, è uno dei pilastri dell'approccio DevOps, che vede la convergenza tra sviluppatori e addetti alle "operations" IT.

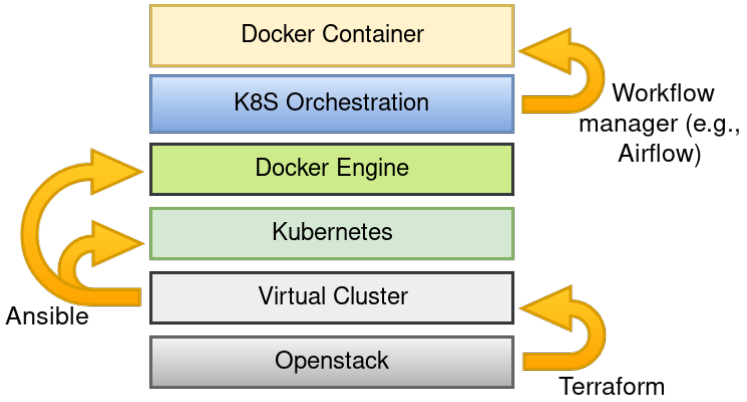

Figura 4.1: Architecture as code stack

L'obiettivo dell'approccio basato su Infrastructure as Code è quello di rendere il più possibile semplice e replicabile il deployment dell'infrastruttura tramite la codifica di ogni aspetto della sua configurazione (e.g., creazione macchine virtuali, installazione e configurazione di software di base, configurazione di rete, etc.). E di fondamentale importanza per garantire la replicabilità della piattaforma software è l'utilizzo della tecnologia dei container come mezzo di distribuzione ed esecuzione di ogni sua componente (sezione 4.1). Applicazioni complesse interamente containerizzate richiedono un adeguato supporto per gestire il coordinamento (*orchestrazione*) delle loro componenti e Kubernetes (sezione 4.2) è considerato lo standard de facto capace di offrire tale supporto. La replicabilità della sua complessa installazione e configurazione è completamente affidata a tool che automatizzano ogni sua parte (sezione 4.3),

così come la creazione dell'infrastruttura (i.e., nodi, rete, storage, etc.) su cui è installato è interamente gestita in modo programmatico (sezione 4.4).

4.1 Containers

I container sono dei pacchetti contenenti software e le relative dipendenze che, grazie alla virtualizzazione a livello di sistema operativo, possono essere facilmente eseguiti e trasportati da un sistema ad un altro. La suddetta virtualizzazione, a cui è associato un overhead nullo o molto basso, prevede l'esistenza di più spazi utente tra loro isolati, detti appunto container. I programmi che girano all'interno di un container hanno accesso solo ai contenuti e alla risorse assegnate a quel container.

Esistono diverse soluzioni per la containerizzazione, tra cui:

- Docker
- RKT
- Singularity
- lmctfy
- LXC

La più usata è indubbiamente Docker[1], che è stata adottata nel progetto TDM.

In Docker, la descrizione dei container è detta immagine ed è espressa tramite una apposita sintassi in file chiamati Dockerfile. A partire dalle immagini è possibile instanziare i container, che possono condividere lo storage con la macchina host e che dialogano all'interno di una sotto rete locale.

Docker fornisce anche una piattaforma per il versionamento e la convisione delle immagini, chiamato Docker Hub, oltre che a una serie di strumenti per l'orchestrazione dei container come Docker Compose e Docker Swarm.

4.2 Orchestrazione di containers: Kubernetes

Kubernetes[2] è un sistema open source, standard de facto, per l'orchestrazione di software basato su container. La piattaforma supporta l'automazione della moltitudine di attività richieste dalla gestione di servizi informatici, come il deployment, il monitoraggio e lo scaling. Inizialmente rilasciato da Google, è ora gestito dalla Cloud Native Computing Foundation. Essendo una piattaforma aperta, non legata ad alcun singolo venditore, Kubernetes (abbreviato *k8s* inoltre consente la portabilità di applicazioni tra diversi fornitori di infrastrutture.

A livello architetturale, un cluster k8s è composto da uno o più nodi master, che formano il *control plane*, e da un insieme di nodi *worker* (vedi Fig. 4.2). Il control plane si occupa della gestione del cluster. Multipli nodi master possono essere usati per fornire un'elevata disponibilità e scalare a un maggior numero di nodi di lavoratori.

[1] https://www.docker.com/
[2] https://kubernetes.io/

D'altro canto, i nodi worker forniscono le risorse di calcolo per l'esecuzione dei container. I carichi di lavoro vengono eseguite in unità chiamate *pods*, che eseguono container strettamente accoppiati che condividono risorse, compreso un unico indirizzo IP. Questa è la risorsa più semplice eseguibile su Kubernetes. I principali servizi di Kubernetes comunicare direttamente attraverso la rete del cluster, mentre tutti i pod sono collegati direttamente attraverso una *virtual overlay network* che esiste solo all'interno del cluster. Questo design permette a tutti i pod all'interno del cluster k8s di indirizzarsi direttamente senza NAT, anche con elevante quantità di pod in esecuzione.

Kubernetes definisce una serie di risorse o componenti quali Pod, Deployment, Service ecc. che consentono l'esecuzione, lo scaling, la persistenza e l'interfacciamento a container software.

Per implementare la rete virtuale Kubernetes può usare uno qualsiasi dei seguenti plugin compatibile con la Container Network Interface (CNI)[3]. Per gli scopi di questo lavoro, gli autori usano Flannel[4]. Flannel assegna una sottorete ad ogni host che partecipa alla rete overlay da cui possono essere assegnati gli indirizzi IP dei pod [1]. I daemon di Flannel girano su ogni nodo del cluster Kubernetes e su ognuno implementano un dispositivo di rete virtuale TUN per ogni pod che gira sul nodo. In una tipica configurazione basata su Flannel, il traffico IP tra i pod sullo stesso nodo viene instradato direttamente attraverso un bridge – dispositivo di rete gestito dal motore dei container (e.g., Docker). Invece, il traffico tra pod su nodi diversi è incapsulato da Flannel sul nodo del pod e trasmesso al nodo di destinazione attraverso una convenzionale Linux VXLAN – che racchiude il traffico in datagrammi UDP; alla destinazione il daemon Flannel corrispondente interpreta il pacchetto e lo fornisce al dispositivo di rete overlay indirizzato e collegato al pod attraverso il bridge device creato dal motore dei container. Flannel può anche essere configurato per utilizzare metodi diversi dalla VXLAN per la trasmissione dei pacchetti encapsulati – ad esempio, esistono plugin specifici per il cloud forniti da fornitori IaaS – ma questi non rientrano nell'ambito di questo lavoro.

4.2.1 Componenti Kubernetes

Le componenti Kubernetes (*resources*) sono definite, secondo un approccio *dichiarativo*, all'interno di file yaml o json che ne specificano lo stato desiderato.

L'unità di deployment su Kubernetes è il `Pod`, il quale si colloca ad un livello di astrazione superiore rispetto a quello dei container. Esso, infatti, consente di raggruppare uno o più container in un unico contesto di esecuzione, definendo l'insieme di risorse (e.g., cpu, rete, storage, etc.) che tali container andranno a condividere.

Al di sopra della risorsa di tipo `Pod` ci sono degli ulteriori livelli di astrazione, tra cui i particolarmente utilizzati `Deployment` e `StatefulSet`. Entrambi definiscono delle policy per il riavvio (*restart*) dei container e consentono di gestire lo scaling e l'aggiornamento dei servizi che in essi vengono eseguiti senza soluzione di continuità. In particolare, mentre i `Deployment` sono utilizzati per definire servizi eseguiti su una o più repliche indistinguibili di una stessa specifica di Pod, gli `StatefulSet` conferiscono un'identità univoca a ciascun

[3] https://github.com/containernetworking/cni
[4] https://github.com/coreos/flannel

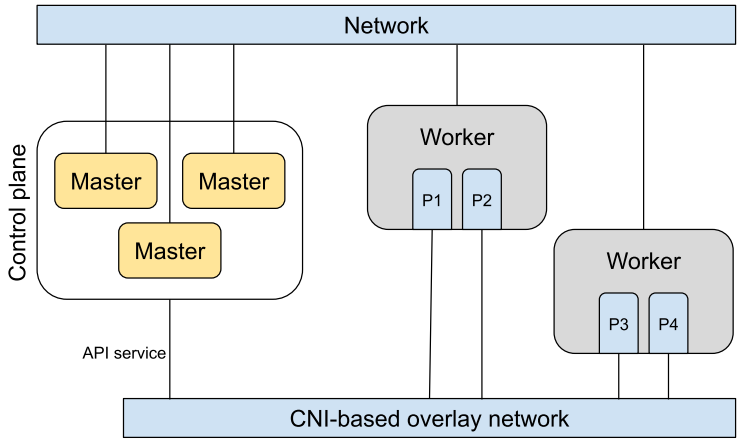

Figura 4.2: Architettura ad alto livello di Kubernetes. I componenti marcati "Pn" sono pods che stanno girando su nodi worker.

Pod del set. Pur generati a partire da una stessa specifica di Pod, questi non sono intercambiabili e mantengono tale loro identità anche in caso di eventuali riavvii. Attraverso risorse di tipo `Persistence Volume Claim` è possibile collegare dei volumi all'interno dei Pod, in modo da offrir loro supporto per la persistenza di dati.

La risorsa `Service`, infine, consente di esporre i servizi implementati da uno o più Pod all'interno del cluster ad altri Pod dello stesso cluster (`ClusterIP service`) o a componenti software ad esso esterni (e.g., `NodePort`, `LoadBalancer service`).

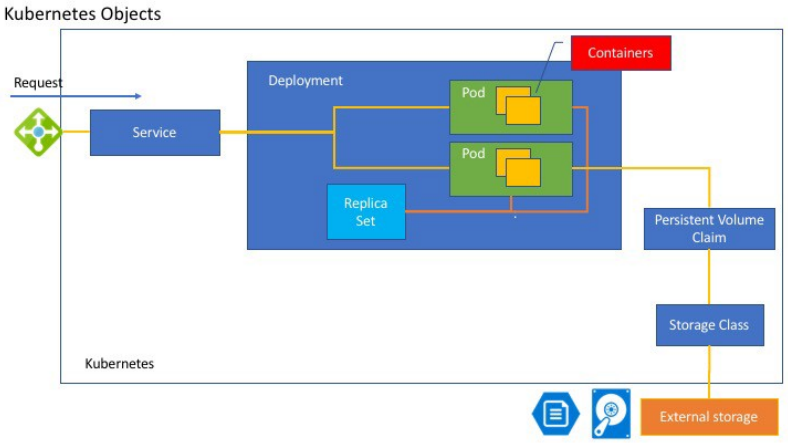

Figura 4.3: Principali componenti del modello applicativo fornito da Kuberneteso (immagine di Tsuyoshi Ushio).

4.2.2 Helm: l'installazione di applicazioni su K8S

L'installazione dei componenti applicativi su Kubernetes avviene, come detto in precedenza, tramite file di configurazione. Il progetto Helm[5] fornisce uno strumento open source che facilita la loro installazione e l'aggiornamento, nonché la condivisione.

Il componente base di Helm è la Chart, una serie di template che, una volta opportunamente valorizzati, definiscono una specifica instanza di risorse Kubernetes e quindi del servizio o applicazione associati. Le Chart sono strumenti flessibili e facilmente condivisibili. Sono disponibili diversi repository dai quali è possibile scaricare le chart.

Le principali funzionalità offerte da Helm sono:

- creazione *from scratch* delle chart
- pacchettizzazione delle chart
- creazione e interazione con i repository
- installazione e disinstallazione delle chart
- gestione del ciclo di release delle chart

L'architettura di Helm è di tipo client/server: localmente l'utente utilizza il cliente per la gestione delle chart, dialogando con il server chiamato *Tiller* che viene installato come servizio Kubernetes con il comando *helm init*.

Nell'ambito del progetto TDM, sono state scritte ex novo o modificate le chart per il software utilizzato. Le chart sono disponibili su un repository pubblico[6].

4.3 Deployment di Kubernetes

Kubernetes è un sistema complesso, costituito da una molteplicità di componenti funzionali la cui installazione e configurazione risulta, in generale, difficile e non facilmente riproducibile.

I nodi computazionali prescelti all'esecuzione di Kubernetes devono soddisfare diversi preliminari requisiti che vanno dalla disponibilità di alcuni componenti software di base opportunamente configurati (e.g., `docker, kubeadm, kubectl`), al precarimento di particolari moduli kernel, configurazione di regole di instradamento, etc. A queste e molte altre azioni richieste per la predispozione dei nodi si accompagnano altrettante numerose e spesso complesse attività di installazione e configurazione delle componenti di Kubernetes. Per superare tale complessità e, al contempo, rendere l'intero processo di installazione riproducibile, sono state proposte varie soluzioni capaci di automatizzarne ogni aspetto. Molte di queste hanno come base comune `Ansible`, un tool open source che consente di automatizzare le procedure di configurazione e gestione di sistemi linux. Il modello di funzionamento di Ansible prevede l'utilizzo di *nodi controller* che svolgono il ruolo di orchestratori dell'intero processo di configurazione, eseguendo comandi su gli altri nodi coinvolti nel processo di configurazione. A differenza di altre soluzioni simili, Ansible è completamente *agentless*: esso, infatti, necessità

[5] https://helm.sh/
[6] https://github.com/crs4/helm-charts

unicamente di poter accedere ai nodi via SSH e non richiede l'esecuzione di alcun ulteriore software.

Nell'ambito del progetto TDM si è optato per l'utilizzo di `KubeSpray` [2], un progetto portato avanti dalla stessa comunità impegnata nello sviluppo di Kubernetes. Esso si configura come una collezione di moduli Ansible (*playbook*) che rendono facilmente configurabile e automatizzabile l'intero processo di installazione di Kubernetes su un insieme di nodi preliminarmente istanziati.

4.4 Creazione automatizzata dell'infrastruttura

Come previsto dal paradigma Infrastructure as Code, l'infrastruttura deve essere creata e configurata in modo programmatico. I tool più utilizzati a tale scopo sono CloudFormation and Terraform[7]. Mentre il primo è specifico per l'ambiente AWS, il secondo supporta i principali provider di IaaS (i.e., AWS, OpenStack, Azure, etc.). In particolare, la sua compatibilità con OpenStack lo rende il candidato più adatto ad essere utilizzato nello specifico contesto del progetto TDM.

Terraform consente la creazione, la modifica ed il versionamento di un'infrastruttura in modo sicuro ed efficiente. Si fonda su un approccio dichiarativo alla definizione dell'infrastruttura, ovvero prevede che di quest'ultima venga descritta la configurazione desiderata – in luogo dei passi necessari per ottenerla, come accade negli approcci procedurali. A partire dalla descrizione dell'infrastruttura, Terraform genera inizialmente il piano di azioni (*execution plan*) necessarie alla sua istanziazione, per poi procedere alla loro esecuzione e portare a compimento l'effettiva costruzione dell'infrastruttura. Le risorse create vengono tracciate in un grafo (*Resouce Graph*) che riflette lo stato attuale di configurazione dell'infrastruttura e rende possibile modifiche incrementali ogniqualvolta risulti necessario. La varie informazioni di stato mantenute da Terraform vengono salvate all'interno di file `.tfstate`.

La definizione dell'infrastruttura avviene tramite file testuali in cui vengono combinati specifici moduli capaci di gestire le varie risorse di interesse per comporre l'infrastruttura (e.g., istanze di macchine (*compute instances*, storage, elementi di rete, etc. A titolo d'esempio, si riporta nel listato 4.1 la definizione Terraform di istanza atta all'esecuzione di un worker node Kubernetes

```
resource "openstack_compute_instance_v2" "k8s_node" {
name        = "${var.cluster_name}-k8s-node-${count.index+1}"
count       = "${var.number_of_k8s_nodes}"
availability_zone = "${element(var.az_list, count.index)}"
image_name = "${var.image}"
flavor_id  = "${var.flavor_k8s_node}"
key_pair   = "${openstack_compute_keypair_v2.k8s.name}"

network {
  name = "${var.network_name}"
```

[7] https://www.terraform.io/

```
  }

  security_groups = ["${openstack_networking_secgroup_v2.k8s.
     name}",
    "${openstack_networking_secgroup_v2.bastion.name}",
    "${openstack_networking_secgroup_v2.worker.name}",
    "default",
  ]

  metadata = {
    ssh_user         = "${var.ssh_user}"
    kubespray_groups = "kube-node,k8s-cluster,${var.
       supplementary_node_groups}"
    depends_on       = "${var.network_id}"
  }

  provisioner "local-exec" {
    command = "sed s/USER/${var.ssh_user}/ contrib/terraform/
       openstack/ansible_bastion_template.txt | sed s/
       BASTION_ADDRESS/${element( concat(var.bastion_fips,
       var.k8s_node_fips), 0)}/ > contrib/terraform/
       group_vars/no-floating.yml"
  }

  scheduler_hints {
    group  = "${openstack_compute_servergroup_v2.node_sg.id}"
  }
}
```

Listing 4.1: *Definizione Terraform di una* `compute instance` *deputata all'esecuzione di un worker node Kubernetes.*

I principali comandi Terraform utili alla gestione di un'infrastruttura sono:

- `init`: predispone l'ambiente locale per l'esecuzione di Terraform;
- `apply`: calcola ed applica un piano di azioni per istanziare l'infrastruttura desiderata;
- `destroy`: distrugge le risorse create nell'ultima esecuzione del comando `apply`.

4.5 Il tool manage-cluster

I tool che automatizzano l'installazione di Kubernetes (sezione 4.3) e la creazione dell'infrastruttura (sezione 4.4) costituiscono un prerequisito che deve essere soddisfatto dall'ambiente sul quale vengono eseguiti. Al fine di eliminarne da qui la dipendenza, è stato realizzato il tool

`manage-cluster` [3], il quale incapsula ogni requisito software all'interno di un'immagine Docker e ne semplifica l'utilizzo attraverso uno script bash.

La funzionalità principale del tool `manager-cluster` è quella di eseguire un deployment di Kubernetes su risorse OpenStack. In particolare, esso fa uso di Terraform per la creazione dell'infrastruttura di macchine virtuali su cui, mediante KubeSpray, esegue l'installazione e configurazione di un cluster Kubernetes. Sia Terraform che KubeSpray sono inclusi nell'immagine Docker su cui `manage-cluster` si basa e che viene trasparentemente caricata dallo script di gestione. Nel dettaglio, la versione di KubeSpray utilizzata è quella del fork realizzato nell'ambito del progetto, la quale contiene alcune modifiche per gestire la specifica configurazione necessaria per il progetto.

Come mostrato nel listato 4.2, `manage-cluster` offre una semplifice interfaccia da linea di comando (CLI) che espone le principali funzionalità di cui si ha bisogno per istanziare un cluster Kubernetes a partire da un set di risorse OpenStack:

```
Usage of 'manage-cluster'

manage-cluster <COMMAND> <CLUSTER_DIR>
manage-cluster -h        prints this help message
manage-cluster -v        prints the 'manage-cluster' version

COMMAND:
  template       creates a template cluster configuration
     directory
  deploy         creates virtual machines
  deploy-k8s     deploys kubernetes
  config-cluster customize cluster with CRS4-specific
     configuration
  destroy        destroys virtual machines
  config-client  configures kubectl
  get-master-ips prints out master IPs for the cluster,
                 one per line (cluster must be deployed)
  shell          opens a shell in the manage-cluster
     container

CLUSTER_DIR:
  Path to the directory containing the cluster's
     configuration
  (i.e., terraform files and artifacts)
```

Listing 4.2: *Interfaccia da linea di comando di* `manage-cluster`

La configurazione del cluster da istanziare è definita tramite una serie di file configurazione, di possono essere preliminarmente inizializzati tramite il comando `template`. Tra i vari file

generati come output, vi è il file `cluster.tf`, che consente di definire numero e tipo di nodi da utilizzare per il deploy delle componenti del cluster Kubernetes.

```
cluster_name = "tdm-stage"
dns_nameservers = ["172.30.3.211", "172.30.3.212"]
# SSH key to use for access to nodes
public_key_path = "../artifacts/tdm-stage-ssh-key.pub"

# image to use for nodes
image = "ubuntu-18.04"
# user on the node
ssh_user = "ubuntu"

# 0|1 bastion nodes
number_of_bastions = 0
#flavor_bastion = "<UUID>"
flavor_bastion = "0298dfce-aa0f-45d0-91a6-ca8ac2313d94"
#bastion_allowed_remote_ip = ["0.0.0.0/0"]

# standalone etcds
number_of_etcd = 0

# masters
number_of_k8s_masters = 0
number_of_k8s_masters_ext_net = 3
number_of_k8s_masters_no_etcd = 0
number_of_k8s_masters_no_floating_ip = 0
number_of_k8s_masters_no_floating_ip_no_etcd = 0
flavor_k8s_master = "a64d5bba-5f3f-4c40-83bc-41bca5787341" #
    tdm.m2

# nodes
number_of_k8s_nodes = 0
number_of_k8s_nodes_ext_net = 2
number_of_k8s_nodes_no_floating_ip = 0
flavor_k8s_node = "f92de58b-fa94-4efe-8e70-3b5ca96ed3b4" # tdm
    .worker

# k8s data nodes
number_of_k8s_data_nodes_ext_net = 5
flavor_k8s_data_node = "ad25e075-11f0-412e-8514-d6b4c83b362f"

# networking
network_name = "tdm-stage-net"
external_net = "2f0db58d-fd9f-4cd8-83fb-59c225a06dc0"
```

```
subnet_cidr = "10.99.99.0/24"
floatingip_pool = "external_net"

# scheduling policies
master_vm_scheduler_policy = "soft-anti-affinity"
node_vm_scheduler_policy = "soft-anti-affinity"
```

Listing 4.3: *Esempio di file* `tf/cluster.tf` *per la configurazione del numero e tipo di nodi k8s.*

4.6 Monitoring dell'infrastruttura

La complessità dell'infrastruttura rende spesso difficile avere un'idea chiara del suo stato di funzionamento. I suoi nodi di calcolo, le applicazioni che su questi vengono eseguite e i sottosistemi di rete che consentono ai componenti di dialogare fra loro possono esibire sporadici malfunzionamenti. Di qui la necessità di strumenti capaci di raccogliere e mostrare in tempo reale quale sia lo stato attuale di funzionamento del sistema e di fornire notifiche in caso di eventuali malfunzionamenti. Tali strumenti, infatti, consentono intraprendere opportune azioni atte ad assicurare la continuità di esercizio della piattaforma e dei servizi che attraverso di essa vengono erogati.

I dati necessari ad alimentari l'attività di monitoring dell'infrastruttura sono tipicamente di due tipi:

- *metriche* sul livello di utilizzo di risorse dei componenti di sistema e sulle performance dei componenti software in esecuzione;
- *log* prodotti dai componenti software di sistema e applicativi in esecuzione sulla piattaforma.

Un monitoring efficace richiede che tali dati siano raccolti e memorizzati per poter essere facilmente fruibili e consultabili attraverso interfacce grafiche capaci di fornire una visione sinottica del stato globale di funzionamento del sistema o di dettaglio su specifiche sue componenti. Varie sono le soluzioni commerciali ed open source che rispondono a tali singole necessità e, sovente, vengono distribuite in modo combinato, come *stack* applicativi completi per il monitoring. Nelle successive due sottosezioni verrano illustrate i due stack applicativi open source che si scelto di utilizzare nell'ambito del progetto TDM per il monitoring di sistema attraverso log e metriche.

4.6.1 Monitoring metriche: Prometheus-Grafana

La gestione delle metriche di sistema è affidata a Prometheus e Grafana. Il primo si occupa del recupero e memorizzazione di metriche esposte da componenti software opportunamente "instrumentati" per esporle; il secondo, invece, ne consente la visualizzazione attraverso personalizzabili viste grafiche.

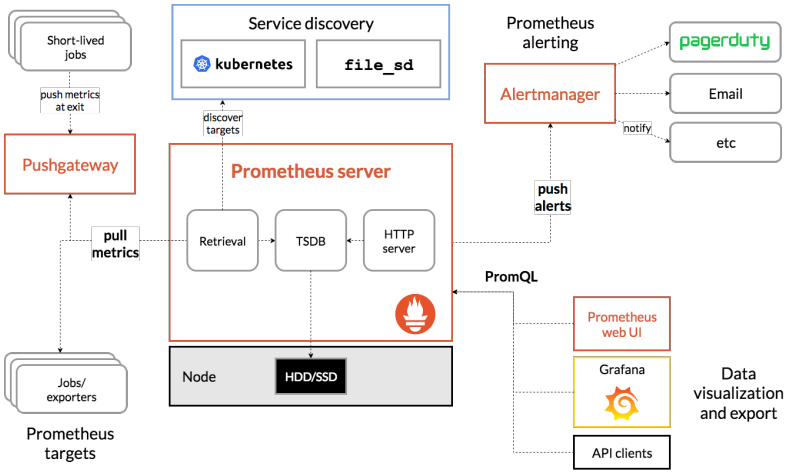

Figura 4.4: Diagramma architetturale di Promethues e dei suoi principali componenti

Promethues. E' un sistema open-source per il monitoring ed l'alerting che conosce un'ampia adozione e vanta una vasta e molto attiva comunità di sviluppatori.

Le principali caratteristiche che esso offre sono:

- un modello dati multi-dimensionale con serie temporali identificate tramite nome di metrica;
- un flessibile linguaggio di interrogazione, `PromQL`;
- un modello *pull* per il recupero delle collezioni di serie temporali su protocollo HTTP;
- gateway intermedi per l'aggregazione ed invio (*push*) di nuove serie temporali;
- registrazione di sorgenti di metriche tramite *autodiscovery* o configurazione statica.

Come illustrato in Fig. 4.4, Prometheus recupera – sia direttamente che attraverso l'intermediazione di gateway (*push gateway*) – le metriche prodotte da job opportunamente instrumentati attraverso le librerie fornite come parte della sua distribuzione. Le sorgenti di metriche (*targets*) possono essere staticamente configurate o identificate tramite funzionalità di autodiscovery – queste ultime consentono, ad esempio, di ottenere metriche da tutti le risorse in esecuzione su Kubernetes senza alcun bisogno di configurazione. Le metriche ottenute vengono, dunque, memorizzate dal Promethues server, dove varie elaborazioni vengono eseguite al fine, ad esempio, di ottenere e memorizzare nuove metriche aggregate o generare degli alert. Tutte le metriche registrate possono essere interrogate utilizzando il linguaggio di query `PromQL` attraverso l'interfaccia web integrata (*Prometheus web UI*) o varie API client.

Grafana. Lo strumento sicuramente più diffuso per fruire delle metriche memorizzate in Promethues è Grafana, un tool open source per la visualizzazione di dati provenienti da varie sorgenti, quali *Graphite, InfluxDB, Promethues,* etc. Grafana consente di interrogare e visualizzare dati provenienti da sorgenti eterogenee, dando la possibilità all'utente di combinarle

in viste anche molto complesse (*dashboard*) che possono essere esplorate e condivise. La Fig. 4.5 mostra un esempio di dashboard che offre una vista globale dello stato del cluster Kubernetes in termini di risorse utilizzate (e.g., cpu, memoria, storage, etc.) e processi (pod) in esecuzione.

Figura 4.5: Overview dello stato del cluster Kubernetes, rappresentato mediante dashboard Grafana

4.6.2 Monitoring log: Fluentd-ElasticSearch-Kibana

L'uso combinato dei tool Fluentd, ElasticSearch e Kibana consente di gestire di efficacemente i log prodotti dalle varie componenti software in esecuzione sulla piattaforma TDM.

In particolare, Fluentd è il componente che raccoglie gli output (da stdout e stderr) prodotti dalle eterogenee applicazioni in esecuzione sulla piattaforma e li omogeneizzare in un formato JSON condiviso che ne semplifica la memorizzazione e fruizione. I log così prodotti vengono, quindi, memorizzati in ElasticSearch che, indicizzandoli, ne consente un'efficiente interrogazione. Kibana, infine, dotati di opportuni connettori verso ElasticSearch, ne semplifica la ricerca e visualizzazione attraverso un'intuitica interfaccia grafica web. La Fig. 4.6 mostra un esempio di come output prodotti da software in esecuzione sul cluster Kubernetes vengano convertiti in messaggi di log dal formato standard, così da poter essere facilmente memorizzati, interrogati e visualizzati.

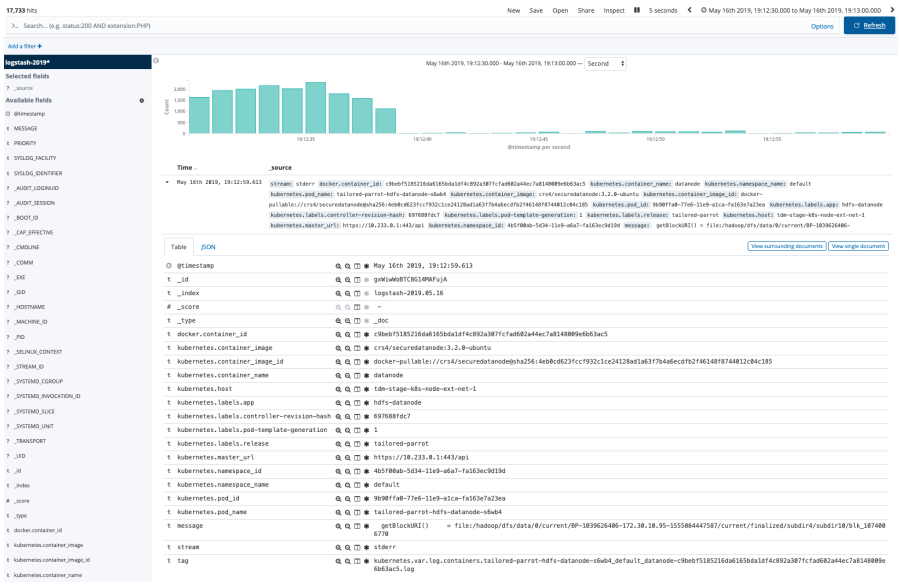

Figura 4.6: Monitoring dei log prodotti dalle componenti software in Kubernetes

5 TDM system components

In questo capitolo descriviamo come sono implementate praticamente le varie componenti del sistema TDM. Per ognuna di queste componenti viene prima data una descrizione generale delle sue funzionalità, dettagli sulla sua implementazione e, in fine su come è stato pacchettizzato per gestire il suo deploy sotto kubernetes. Nello specifico, verranno discusse le seguenti macro componenti:

- il sistema di gestione dei dati, organizzato come un data cube che si appoggia ad un data lake;
- il sistema di accesso alle risorse di calcolo, organizzato per gestire sia attività computazionali gestite da workflows che un accesso interattivo per l'esplorazione dei risultati.

5.1 Data lake

In questa sezione vengono descritti, nell'ordine seguente,

- il sistema scalabile di storage alla base di tutto il sotto sistema di Data lake;
- la strategia di indicizzazione spazio-temporale utilizzata.

5.1.1 Sistema di stoccaggio scalabile dei dati

I grandi volumi di dati eterogenei trattati nell'ambito del progetto richiedono l'utilizzo di un file system scalabile, robusto e ad alto throughput. I dati, inoltre, una volta acquisiti e memorizzati, non vengono più modificati: si tratta del modello WORM ("Write Once, Read Many"), per cui sono particolarmente adatti i sistemi che privilegiano l'efficienza in lettura. Per soddisfare queste esigenze, il progetto utilizza HDFS (Hadoop Distributed File System), un file system distribuito open source progettato per l'elevata scalabilità, l'efficienza nello streaming (lettura senza interruzioni) di grossi data set e la tolleranza ai guasti hardware.

HDFS ha un'architettura master-slave: un cluster HDFS è composto da un singolo nodo master (due in modalità high availability), il NameNode, che gestisce i metadati e le richieste di accesso da parte dei client, e un numero variabile di nodi slave, i DataNode, che memorizzano i blocchi di dati in cui sono divisi i file. I blocchi, memorizzati come file del file system locale, sono distribuiti con ridondanza, in modo che un guasto isolato non causi perdite di dati. HDFS è accessibile mediante l'API nativa in Java e il suo wrapper in C, un'interfaccia REST, e attraverso le apposite utilities disponibili su riga di comando. Sono disponibili anche interfacce Python di terze parti, ad esempio Pydoop[1], sviluppato dal CRS4.

[1] https://github.com/crs4/pydoop

Dal punto di vista infrastrutturale, un cluster HDFS si realizza mediante l'installazione del codice Java sui nodi del cluster, la configurazione e l'avvio dei servizi NameNode e DataNode.

Nell'ambito del progetto, HDFS è installato automaticamente sul cluster Kubernetes attraverso un apposito Helm chart[2], realizzata dal CRS4 partendo da un precendente lavoro open source. La chart sfrutta delle immagini Docker di Hadoop realizzate dal CRS4[3]. Per i dati gestiti dal NameNode, il chart utilizza i persistent volume (PV) di Kubernetes. Poiché i PV hanno un ciclo di vita indipendente dai servizi che li utilizzano, questo permette ai metadati HDFS di sopravvivere ad un crash o a migrazioni del container che ospita il NameNode tra nodi di calcolo del cluster. Per i blocchi memorizzati dai DataNode, invece, il chart utilizza degli HostPath, directory montate direttamente dal file system del nodo host (la configurazione assicura che ogni datanode insista su un diverso nodo per evitare conflitti nell'accesso al disco).

Nell'architettura TDM, HDFS è il backend di riferimento per i dati multidimensionali di grandi dimensioni come, ad esempio, flussi di immagini radar e satellitari. Una delle operazioni fondamentali sui dati a più dimensioni è lo *slicing*, ossia l'estrazione di sezioni. Ad esempio, l'intensità piovosa in una data area geografica e intervallo temporale può essere rappresentata da un array tridimensionale (due dimensioni spaziali e una temporale). Lo slicing su un dataset di questo tipo consiste nell'estrazione dell'intensità piovosa tra due date assegnate e/o in una sottoarea geografica. L'architettura TDM utilizza TileDB[4], una libreria specializzata nella gestione degli array multidimensionali, per assicurare l'efficienza nell'estrazione di slice. L'integrazione di TileDB nel sistema è agevolata dal supporto per il backend HDFS e dalla disponibilità di un'interfaccia Python, il che rende la libreria utilizzabile in un notebook Jupyter.

5.1.2 Sistema di indicizzazione uniforme

Una delle caratteristiche chiave della piattaforma TDM è l'integrazione di dati georeferenziati di varia natura, largamente divergenti nelle loro caratteristiche. Il sistema deve gestire in maniera uniforme dati compatti e puntiformi, come una singola rilevazione di temperatura ambientale, e dati complessi e impegnativi come la caratterizzazione di un volume atmosferico calcolato in una simulazione meteo. Per raggiungere gli obiettivi della piattaforma si è optato per un'architettura ibrida, che affianca ad un sottosistema di storage adatto ai dati più voluminosi (vedasi ss. 5.1.1) un sistema di indicizzazione centrale, che implementa in maniera prioritaria le interrogazioni per intervallo temporale e posizione geografica, ma supporta anche interrogazioni efficienti sui vari altri metadati.

Il componente che permette l'efficiente interrogazione dei dati e delle sorgenti si basa su un database PostgreSQL con le estensioni specializzate TimescaleDB[5] e PostGIS[6] (entrambe open source). I database relazionali di norma presentano dei punti deboli per quanto riguarda la gestione delle serie temporali. Infatti, numerosi database a colonna sono stati creati per la

[2] https://github.com/tdm-project/kubehdfs
[3] https://github.com/crs4/hadoop-docker
[4] https://github.com/TileDB-Inc/TileDB
[5] https://github.com/timescale/timescaledb
[6] https://github.com/postgis/postgis

gestione di questo tipo di dato, sacrificando i vantaggi del modello relazionale dei dati a favore dell'efficiente gestione di questo caso specifico (e.g., Prometheus, InfluxDB). TimescaleDB introduce delle funzionalità avanzate per la gestione efficiente delle serie temporali in PostgreSQL, quindi senza abbandonare il modello relazionale e raggiungendo delle prestazioni competitive con gli altri approcci. La possibilità di utilizzare PostgreSQL per questa applicazione permette di utilizzare PostGIS, che rappresenta l'attuale stato dell'arte degli strumenti open source disponibili per l'indicizzazione di dati geolocalizzati. Inoltre, utilizzare un gestore di database largamente diffuso come PostgreSQL ha il vantaggio di essere automaticamente compatibili con numerosi strumenti esistenti: e.g., immagini Docker, Helm chart, SQLalchemy e Alembic, ecc.

5.1.2.1 Gestione versionata dello schema

Lo schema del RDBMS di indicizzazione è stato studiato per raggiungere un buon compromesso tra prestazioni in presenza di inserimenti e interrogazioni concorrenti, e flessibilità nelle fonti e tipi di dati temporali e georeferenziati da gestire. Tuttavia, è necessario prevedere delle modifiche in futuro allo schema del database per accomodare future esigenze. Per gestire le modifiche in maniera ordinata abbiamo integrato nell'immagine Docker di TimescaleDB un nuovo modulo che, all'avvio e se richiesto, verifica la versione del DB presente su disco e la aggiorna all'ultima versione applicando gli opportuni comandi SQL. Il modulo è implementato utilizzando il software open source Alembic[7].

5.1.2.2 Installazione su Kubernetes

Come tutti i componenti della piattaforma TDM, anche il database di indicizzazione viene installato su Kubernetes attraverso un Helm chart. Il chart di questo componente è una versione personalizzata del chart ufficiale per PostgreSQL prodotto da Bitnami. In particolare, la chart è stata modificata per aggiungere il supporto alle nuove funzionalità per la gestione dello schema del database. Nella configurazione attuale, il database utilizza storage fornito dal cluster OpenStack attraverso Cinder; a basso livello l'accesso allo storage è quindi effettuato tramite NFS e viene usato un sistema di storage generico. Tuttavia l'approccio mostra prestazioni più che sufficienti per i requisiti (vedasi gli esperimenti descritti in sez. 7.1.2) e mantiene tutta la robustezza e flessibilità fornita dall'utilizzare un cloud-native storage ben integrato con Kubernetes.

5.1.2.3 API sistema indicizzazione

Dal punto di vista logico, i dati del progetto possono essere strutturati secondo una dimensione temporale ed una o più dimensioni spaziali. L'interrogazione (query) dei dati stessi, quindi, deve consentire la selezione di sottoinsiemi individuati da più limiti spaziali e temporali. A ciò si aggiunge la selezione delle sorgenti in base ad attributi quali il modello e le proprietà misurate.

[7] https://alembic.sqlalchemy.org

L'interfaccia di accesso alle risorse TDM è realizzata mediante un servizio web di tipo REST (REpresentational State Transfer). Le entità principali modellate sono "sensor" (sorgente di dati), "sensor_type" (tipologia di sorgente, ad esempio un sensore di temperatura) e "measure" (dato numerico rilevato dal sensore). Ad ogni sensor sono associati un sensor_type e una geometria geograficamente localizzata, mentre ad ogni measure sono associati il sensore che ha effettuato la rilevazione e l'istante di tempo in cui è avvenuta.

Le query si eseguono tramite opportune chiamate HTTP al servizio web. Ad esempio, supponendo che l'indirizzo di base sia http://api.tdm-project.it, una chiamata al seguente indirizzo permette di ottenere la lista di tutti i sensor_type:

```
http://api.tdm-project.it/sensor_types
```

I record vengono restituiti in formato JSON:

```
    {
        "code": "0fd67c67-c9be-45c6-9719-4c4eada4be65",
        "type": "TemperatureSensor",
        "modelName": "TSensorPro",
        "manufacturerName": "Acme",
        "controlledProperty": ["temperature"]
        [...]
    },
    [...]
```

I sensor_type possono essere filtrati in base a un qualsiasi attributo. Ad esempio (nel seguito, per semplicità, ometteremo l'indirizzo di base):

```
/sensor_types?controlledProperty=temperature
```

Anche i sensori possono essere recuperati tutti insieme o filtrati. La seguente chiamata richiede l'intera lista:

```
GET /sensors

    {"code": "0fd67c67-c9be-45c6-9719-4c4eada4becc",
     "stypecode": "0fd67c67-c9be-45c6-9719-4c4eada4be65",
     "geometry": {"type": "Point", "coordinates": [9.13, 39.22]}
    },
    [...]
```

Richiesta dei sensori di un tipo assegnato (tramite l'attributo "code" del relativo sensor_type):

```
GET /sensors?type=0fd67c67-c9be-45c6-9719-4c4eada4be65
```

Sensori compresi in una determinata area geografica, per i quali sono disponibili delle misure comprese tra due istanti di tempo assegnati (URL non codificato):

```
GET /sensors?footprint=circle((9.22, 30.0), 1000)
            &after=2019-05-02T11:00:00Z
            &before=2019-05-02T11:50:25Z
```

Serie temporale di misure per un dato sensore (URL non codificato):

```
GET /sensors/0fd67c67-c9be-45c6-9719-4c4eada4becc/
    timeseries?after=2019-02-21T11:03:25Z
              &before=2019-05-02T11:50:25Z
```

Il risultato è un oggetto JSON contenente l'istante iniziale ("timebase"), le misure comprese nell'intervallo ("data") e i relativi delta temporali ("timedelta"):

```
{"timebase": "2019-02-21T11:03:25Z",
 "timedelta": [0.11, 0.22, 0.33, 0.44],
 "data": [12000, 12100, 12200, 12300]}
```

L'API REST può essere utilizzata per l'accesso ai dati da parte di altri componenti del sistema, in particolare JupyterHub (illustrato più avanti). Ad esempio:

```
import requests
requests.get("http://api.tdm-project.it/sensors").json()
```

Dal punto di vista architetturale, il servizio query è mediante un'applicazione Flask[8], servita tramite Gunicorn[9]. L'applicazione si appoggia su due componenti fondamentali: il servizio di indicizzazione e il servizio di storage scalabile, entrambi descritti meglio nelle rispettive sezioni precedenti. Come descritto sotto, tutti i componenti sono installati su Kubernetes tramite Helm e opportuni container Docker.

5.2 Accesso alle risorse computazionali

L'accesso alle risorse computazionali è, sostanzialmente, ottimizzato per gestire i seguenti scenari tipici:

- l'acquisizione di dati provenienti dal flusso di messaggi provenienti dalla rete di sensori TDM;
- l'esecuzione di workflow computazionali per il recupero ed il caricamento nel data lake di dati provenienti da sorgenti esterne, come, ad esempio, il radar meterologico cittadino ed il mosaico della rete nazionale dei radar meteorologici gestiti dalla protezione civile,

[8] http://flask.pocoo.org
[9] https://gunicorn.org

le immagini dei satelliti Sentinel 1 e 2 di Copernicus, i dati di simulazione meteorologica a scala planetaria forniti dal NOAA ed utilizzati per l'inizializzazione delle simulazioni meteo a scala locale.

- l'esplorazione interattiva dei dati presenti nel data lake attraverso notebooks jupyter.

In alcuni casi, ad esempio l'esecuzione delle simulazioni meteo, si utilizzano le risorse computazionali del cluster HPC. Per rendere omogeno l'utilizzo di quest'ultimo rimanendo all'interno di un approccio basato su Kubernetes si è realizzato un nuovo sistema per l'integrazione dinamica in cluster Kubernetes di nodi gestiti da sistemi DRM come SGE [4].

5.2.1 Gestione Workflow

L'avvio e la gestione delle diverse pipeline di elaborazione batch è svolta dal Workflow Manager *Apache Airflow*. Airlfow, partendo da un grafo che descrive la pipeline in termini di operazioni ed eventi, avvia i diversi task, controlla la loro esecuzione e determina le azioni da intraprendere al verificarsi di eventi quali l'uscita del task con successo o con errore. I file che contengono le informazioni su come avviare e gestire le pipeline sono chiamati *DAG*.

Il deployment usato per TDM prevede per l'applicazione Airflow quattro microservizi, tutti distribuiti come container Docker:

- Airflow-WebServer, la web dashboard/gui con la quale si gestiscono interattivamente le pipeline;
- Airflow-Scheduler, il componente che partendo dai DAG, avvia le pipeline in determinati orari o a determinati intervalli;
- Airflow-Worker, il componente che esegue i task delle pipeline;
- PostgreSQL, il database a cui Airflow fa riferimento per lo scheduling, logging e statistica di funzionamento.

L'implementazione TDM del sistema Airflow, in particolare, fa utilizzo dell'operatore *Kubernetes* per l'esecuzione dei task. Quando una pipeline viene avviata, il nodo Worker, leggendo il relativo DAG, avvia uno o più container Docker sull'infrastruttura Kubernetes, distribuendo i task in maniera dinamica sui nodi che hanno adeguate risorse disponibili. Questo a sua volta fa si che solo un nodo Worker rimanga costantemente allocato e che l'esecuzione di elaborazioni anche intensive ma periodiche e limitate nel tempo possano, all'occorrenza, sfruttare tutte le risorse disponibili rilasciandole quando non più necessarie.

Per distribuire i DAG ai vari microservizi del sistema Airflow è stato inserito nell'infrastruttura un *provider di storage NFS*. Il filesystem NFS è un filesystem condiviso che permette la distribuzione di *share* o alberi a diversi nodi gestendone le politiche di accesso, permesso e aggiornamento. I DAG copiati sullo share sono quindi resi disponibili contemporaneamente ai componenti Web, Scheduler e Worker. Nel listato 5.1 è riportato un estratto di un DAG per Airflow che avvia la copia dei dati dal Radar Meteorologico verso lo storage HDFS di TDM per la successiva elaborazione. Il DAG è scritto in linguaggio Python. L'oggetto *DAG* rappresenta la pipeline e ne fornisce i settings generali, come scheduling, notifiche e retries. Il task da eseguire è descritto dall'oggetto *KubernetesPodOperator*. I parametri di quest'ultimo sono parametri tipici di un POD Kubernetes, come volumi, mount, container e immagini.

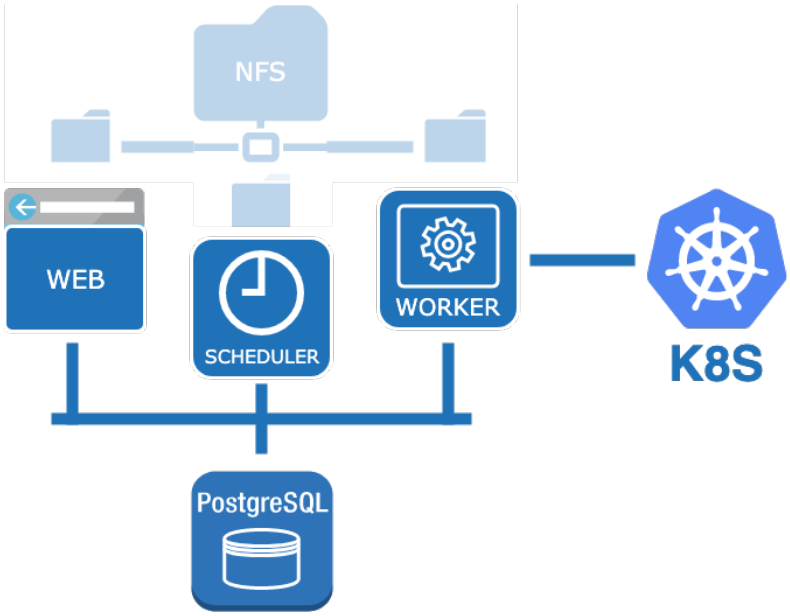

Figura 5.1: Microservizi dell'applicazione Airflow

Listing 5.1: *Esempio di DAG Airflow*

```
volume_mount = VolumeMount(
    'scripts',
    mount_path='/scripts/',
    sub_path=None,
    read_only=True)

volume_config = {
    'persistentVolumeClaim':
    {
        'claimName': 'airflow-scripts',
    }
}

scripts_volume = Volume(name='scripts', configs=volume_config)


default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime.utcnow(),
```

```
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 0,
    'retry_delay': timedelta(minutes=5)
}

dag = DAG(
    'radar_sync_dag',
    default_args=default_args,
    schedule_interval=timedelta(hours=12)
)


pydoop = KubernetesPodOperator(
    namespace='default',
    image="crs4/pydoop-client",
    labels={"cluster": "tdm"},
    name="sync-task",
    task_id="sync-task",
    get_logs=True,
    is_delete_operator_pod=True,
    volumes=[scripts_volume],
    volume_mounts=[volume_mount],
    dag=dag
)
```

5.2.2 Supporto all'esplorazione interattiva

L'accesso ai dati in maniera complessa e programmatica è supportato tramite Jupyter Notebook[10], una applicazione Web open source per la creazione, la condivisione e l'esecuzione di codice sorgente. Le principali caratteristiche di Jupyter Notebook sono:

- editing del codice sul browser con syntax highlighting, indentazione, completamento tramite tab completion;
- esecuzione del codice sul browser, con la visualizzazione dell'output ad esso associato;
- supporto per oltre 40 linguaggi di programmazione, tra cui Python, R, Julia, and Scala;
- possibilità di condividere il codice;
- visualizzazione dell'esito della computazione tramite molteplici formati tra cui HTML, LaTeX, PNG, SVG;

Alcuni tipici utilizzi sono: manipolazioni di dati, simulazioni numeriche, modellazioni statistiche, visualizzazione, machine learning.

[10] https://jupyter.org/

Al fine di consentire a l'esecuzione di Jupyter Notebook in contesto multi utente, in TDM è stato utilizzato JupyterHub[11], che gestisce centralmente l'esecuzione di più server single user di Jupyter Notebook.

JupyterHub fornisce una serie di immagini Docker che possono essere utilizzate per istanziare i Notebook, disponibili sia su GitHub[12] che su DockerHub[13]. Le immagini si differenziano per i pacchetti installati e quindi disponibili all'utente: si va dall'immagine minimale a quella per il machine learning con Tensorflow, a quella per i data scientist che include librerie per l'analisi dei dati in Python, R e Julia, a quella per il calcolo distribuito con il supporto di Spark per Python, R e Scala. Queste immagini possono essere ulteriormente estese, consentendo l'integrazione di altre librerie necessarie per la particolare installazione.

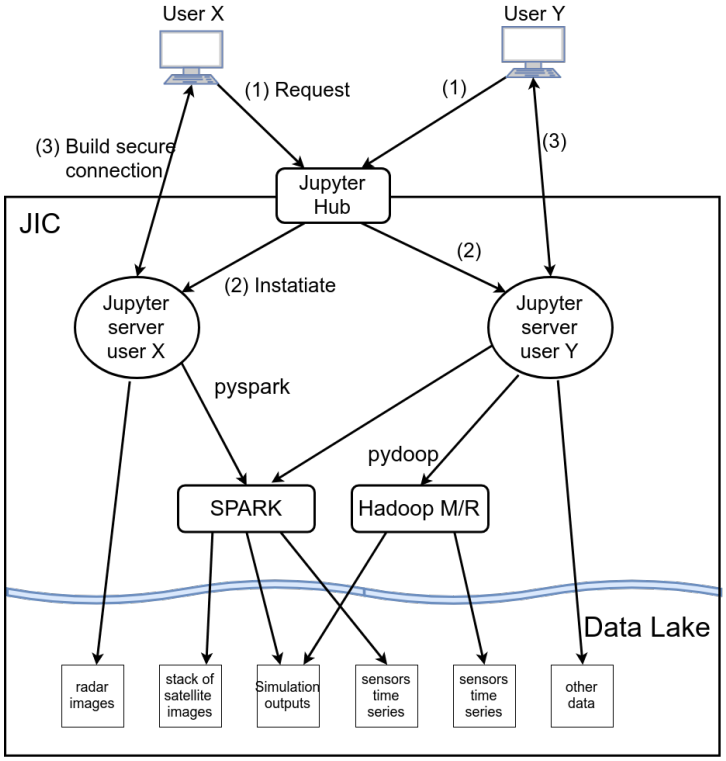

Figura 5.2: Jupyter hub nell'architettura TDM.

A livello architetturale, JupyterHub si pone alla frontiera dell'infrastruttura JIC/OpenStack, consentendo l'accesso a utenti esterni alla rete JIC al Data Lake (figura 5.2). Come gli altri componenti dell'architettura TDM, anche JupyterHub è stato deployato su Kubernetes, utiliz-

[11] https://jupyter.org/hub
[12] https://github.com/jupyter/docker-stacks
[13] https://hub.docker.com/u/jupyter

```
In [ ]: !pip3 install numpy tifffile --user

In [1]: import requests
        import json
        import numpy as np
        import tifffile as tiff
        import math
        import tempfile
        import os
        import tiledb
        import uuid

        from datetime import datetime, timedelta
```

Figura 5.3: Installazione persistente di pacchetti di software all'interno del Notebook.

Figura 5.4: Query su Notebook per il recupero dei sensori in una data zona e in un dato intervallo temporale (sx) e delle serie temporali di un dato sensore in una dato range di date (dx), con relative visualizzazione su una mappa interattiva.

zando la chart Helm ufficiale[14]. Al momento del deployment, la chart crea alcune risorse tra cui l'Hub e un proxy per accedere all'Hub stesso dall'esterno del cluster. All'atto della login di un utente autorizzato (effettuata tramite l'LDAP del JIC), l'Hub assegna ad esso un pod che implementa un server Jupyter Notebook.

La chart di JupyterHub garantisce la persistenza del codice e dei dati inseriti dall'utente all'interno del Notebook, grazie alla creazione di un volume persistente Kubernetes che viene collegato alla *home* del pod su cui gira Notebook. Tale funzionalità consente anche l'installazione persistente di ulteriori pacchetti software da parte dall'utente (purché siano installati localmente e non a livello di sistema), come illustrato nella figura 5.3. Ciò significa che ad ogni accesso al Notebook, l'utente potrà continuare il lavoro ritrovando il medesimo ambiente in maniera trasparente.

In TDM è stata creata una immagine Docker ad hoc per consentire l'analisi dei dati immagazzinati ne Data Lake. L'immagine, pubblicata nel DockerHub[15], estende l'immagine mini-

[14] https://jupyterhub.github.io/helm-chart/
[15] https://hub.docker.com/r/crs4/tdm-jupyter-notebook

male fornita da JupyterHub aggiungendo pacchetti tra cui Pydoop, Pandas, Matplotlib Bokeh. Alcuni esempi delle possibili query eseguibili su TDM tramite Jupyter Notebook e relativa visualizzazione dell'output sono riportati nella figura 5.4.

5.2.3 Integrazione di risorse HPC all'interno di un sistema Kubernetes

Le motivazioni per l'adozione di Kubernetes per implementare una complessa piattaforma informatica sono molteplici e ben fondati. Tuttavia, Kubernetes ha una scarsa compatibilità con le risorse di calcolo HPC che spesso si trovano in contesti di ricerca – incluso il CRS4. Allo stesso tempo, Kubernetes da solo non può soddisfare tutte le esigenze della comunità HPC e non è adatto per sostituire un'infrastruttura HPC esistente. In ogni caso, Kubernetes rappresenta uno strumento per permettere i centri HPC di fornire una più ampia gamma di strumenti o per facilitare l'integrazione di dati derivati da servizi (ad esempio, sensori dell'internet degli oggetti) nel lavoro di modellazione e simulazione spesso eseguito su piattaforme HPC [5].

Nel progetto TDM abbiamo implementato un approccio per far girare Kubernetes su risorse di calcolo ibride HPC e OpenStack, che consente di aggiungere e togliere in maniera dinamica dei nodi worker dal cluster secondo la necessità. Questo approccio consente l'ottimizzazione dell'utilizzo delle risorse di calcolo disponibili, minimizza la necessità di nuovi investimenti in risorse di calcolo aggiuntive, e riduce anche la necessità di appoggiarsi a fornitori di IaaS esterni.

L'approccio consiste nell'integrazione o eliminazione dinamica di nodi HPC nel cluster K8S attivo, come nodi worker. L'implementazione supporta il sistema di code Grid Engine (GE) [6], utilizzato al CRS4. L'effetto quindi è il ridimensionamento delle risorse di calcolo disponibili sul cluster K8S. Questa strategia offre una soluzione per i carichi di lavoro orientati ai servizi e ai container, pur avendo un impatto minimo su un'infrastruttura HPC esistente. Inoltre fornisce un mezzo efficace per gestire carichi di lavoro irregolari su K8S, con relativamente brevi picchi di richiesta, avendo a disposizione un eccesso di capacità HPC – senza convertire in maniera permanente l'infrastruttura da approvvigionamento da batch-oriented a cloud.

5.2.3.1 Obiettivi e Architettura dell'Integrazione

Il livello di integrazione tra Kubernetes e Grid Engine che cerchiamo di raggiungere raggiungere è quello di essere in grado di avviare nodi worker del cluster k8s come job Grid Engine. Il nodo worker deve essere creato lanciando un job GE e distrutto cancellando lo stesso job – anche se manuale, questo semplice approccio consente l'automazione futura tramite la creazione di un controller Kubernetes specifico. Tuttavia, la nostra applicazione impone alcuni requisiti particolari:

- il software del k8s worker deve essere eseguito con privilegi di root;
- affidabile procedura di distruzione per assicurare la pulizia del nodo del cluster utilizzato;
- gang scheduling per supportare l'allocazione di blocchi di nodi come un unico gruppo.

Dati questi requisiti, abbiamo scelto di sfruttare le `Parallel Environment (PE)` [7] di Grid Engine. Le Parallel Environment sono un costrutto attraverso il quale GE prepara l'ambiente per un lavoro da eseguire. Da un punto di vista pratico, permette agli amministratori del cluster di configurare uno script di preparazione e uno di pulizia, che sono rispettivamente eseguiti prima e dopo il lavoro, entrambi con privilegi di root. Quindi, la nostra strategia prevede di eseguire l'avvio del nodo worker e la sua connessione al cluster dallo script di preparazione, mentre lo spegnimento e la pulizia viene effettuata dallo script di pulizia del PE. Con questo stratagemma il comando eseguito per il job diventa un banale "`sleep infinity`" che tiene il job in vita. I nodi worker possono essere creati in gruppi attraverso l'esecuzion di un job che richiede più di un nodo (e quindi i nodi saranno configurati solo quando tutte le risorse saranno disponibili). D'altra parte, worker di k8s viene distrutto cancellando il job con l'opzione appropriata del comando GE (`qdel`), usando l'id di lavoro fornito dal comando di sottomissione.

In sintesi, n worker di k8s possono essere creati con il seguente comando Bash:

```
qsub -pe k8s $(( $n * $slots_per_node )) -b y sleep infinity
```

La chiamata al comando `qsub` restituisce un job ID che sarà successivamente usato per liberare il nodo con `qdel`.

In questo esempio, l'utente non fornisce parametri che identificano il cluster k8s a cui unire i nuovi nodi perché supponiamo un cluster singolo sempre attivo – di fatto, un Kubernetes as a Service. Tuttavia, un utente potrebbe passare questi parametri come variabili al comando `qsub` quando sottomette la richiesta.

5.2.3.2 Prerequisiti per il nodo HPC

Vari prerequisiti devono essere soddisfatti dal nodo HPC prima che possa essere usato per i servizi di Kubernetes. Il suo sistema operativo deve essere Linux e i seguenti software devono essere installati:

1. `kubeadm`
2. `kubelet`
3. `docker`

I primi due sono componenti di Kubernetes e devono essere della stessa versione presente sul cluster a cui il nodo si dovrà collegare, i.e., il *Control Plane*. Il terzo è il componente che gestisce il runtime di container. Sebbene k8s ne supporta anche altri, quali *containerd* e *cri-o*, `docker` è attualmente il container runtime attualmente più diffuso ed utilizzato. Il funzionamento di k8s richiedo che esso sia configurato come servizio di sistema (e.g., gestito tramite `systemctl`).

Come ulteriore requisito, il kernel del sistema Linux installato sul nodo deve essere dotato dei moduli di seguito riportati, necessari al corretto funzionamento del sottosistema di rete di k8s: `ip_vs`, `ip_vs_rr`, `ip_vs_sh`, `ip_vs_wrr`, `nf_conntrack_ipv4`, `overlay`.

5.2.3.3 Networking

Kubernetes richiede che tutti i nodi, `master` e `worker`, siano mutuamente riaggiungibili per l'espletamento delle sue interne funzioni di gestione (in Tab. 5.1 si riportano le principali porte utilizzate).

La comunicazione fra nodi del cluster k8s (*inter-cluster communication*) non costituisce un problema in una tipica installazione di Kubernetes su infrastrutture cloud IaaS come OpenStack. In tali scenari, infatti, i nodi possono comunicare direttamente l'uno con l'altro all'interno di una rete privata virtuale dedicata, dove nessuna restrizione è applicata al traffico generato da e diretto a nodi del cluster. Per converso, è estremamente controllato tutto il traffico diretto al cluster, essendo limitato ad un ristretto set di porte su pochi nodi di frontiera dotati di IP raggiungibili dall'esterno (*floating IP*, secondo terminologia di OpenStack).

Nel caso del deployment ibrido che stiamo considerando, parte dei nodi k8s (tra cui quelli che costituiscono il control plane) risiedono su OpenStack mentre i worker node transienti su un cluster HPC. Per assicurare l'interconnessione fra tutti i nodi, si è scelto di utilizzare una *external network* – i.e., rete accessibile dall'esterno del cluster OpenStack – per interconnettere direttamente i nodi del cluster k8s su OpenStack in luogo della più consueta rete privata virtuale. Ciò consente di soddisfare l'importante requisito che richiede ai nodi k8s su OpenStack di essere direttamente accessibili da quelli allocati sul cluster HPC. Al fine di garantire la replicabilità di tale particolare configurazione, sono stati realizzati degli specifici template Terraform, inclusi nel fork del progetto Kubespray [8] direttamente utilizzato dal componente `manage-cluster` [3]

Tabella 5.1: Traffico in ingresso ai nodi Kubernetes (master e worker), incluso quello relativo alla *Flannel overlay network* che gestisce la comunicazione fra Pod in Kubernetes.

Protocol	**P**ort Range	**N**ode type	**P**urpose
TCP	6443	All	Kubernetes API server
TCP	2379-2380	Master Node	etcd server client API
TCP	10250	All	Kubelet API
TCP	10251-10252	Master Node	scheduler and controller manager
TCP	30000-32767	All	NodePort Services
UDP	8472	All	Flannel

5.2.3.4 Accensione e spegnimento di nodi Grid Engine

Il setup dei nodi worker transienti è affidato alla fase di start del Grid Engine Parallel Environment (PE) che, coordinata dal nodo designato master nel PE, viene eseguita su ciascuno dei nodi da allocare. Dopo una preliminare inizializzazione dell'ambiente del nodo (e.g., caricamento di moduli software e kernel necessari), la procedura prevede l'esecuzione del comando `kubeadm join`, il quale inizializza i componenti Kubernetes e li configura per connetterli al Control Plane.

Due sono i parametri richiesti per l'espletamento del join:

- l'endpoint del server che espone l'API k8s;
- il token utilizzato come informazione segreta per la mutua autenticazione dei nodi master e worker.

Il token di autenticazione può essere generato eseguendo il seguente comando su un nodo master:

```
kubeadm token create --print-join-command
```

I token generati tramite la procedura sopra descritta possono essere utilizzati più volte durante il loro periodo di validità. La durata viene stabilita al momento della creazione in funzione dello specifico scenario d'uso: breve, nel caso di cluster con funzione di autoscaling che genera un nuovo token ad ogni successiva allocazione; oppure lunga (al limite infinita), utile nel caso di allocazione manuale contemplata nel nostro scenario implementativo.

Nella nostra procedura di setup di nodi worker transienti dal cluster HPC è utilizzato il metodo di autenticazione worker-master denominato *Token-based discovery with CA pinning*. Esso prevede che, all'inizio del processo di join, il nodo worker recuperi alcune informazioni dal control plane, inclusa la root certificate authority (CA). Stabilita la mutua identità master-worker tramite il token, il worker è in grado di recuperare tutta le informazioni di configurazione di cui ha bisogno per connettersi e dialogare con il Control Plane. `kubelet` viene, quindi, avviato con la configurazione ottenuta e, al suo startup, provvede alla configurazione dell'overlay network: in questa fase, uno o più riavvi del processo `kubelet` possono essere necessari, fino a che esso non sia riuscito a configurare e avviare correttamente tutti i suoi sotto-componenti. Il processo di startup prevede un periodo (configurabile) di attesa prima di ogni tentativo di riavvio.

La procedura di tear down dei nodi worker è affidata alla fase di *stop* del Parallel Environment. Principalmente, essa prevede l'arresto del daemon `kubelet` in esecuzione sul nodo e la cancellazione del suo stato/configurazione tramite esecuzione del comando `kubeadm reset`. Un operazione finale di pulizia è, infine, eseguita per rimuovere dati locali (e.g., file temporanei, immagini di container Docker, etc..) con lo scopo di ripristinare nel nodo fisico lo stato precedente al suo utilizzo.

5.2.3.5 Configurazione del cluster Kubernetes

Sebbene la configurazione generale di Kubernetes esuli dallo scopo del presente documento, si riportano di seguito alcune note che possono risultare d'interesse per deployment ibridi di Kubernetes come quello da noi descritto.

Selezione dei nodi. L'utilizzo di nodi transienti, che possono essere spenti e deallocati all'improvviso – ad esempio, per il raggiungimento dei limiti utilizzo gestiti dallo scheduler batch HPC – può non risultare applicabile a molte applicazioni. Per tale motivo, sembra ragionevole rendere opzionale l'utilizzo dei suddetti nodi, inibendone l'utilizzo in generale e rendendolo possibile solo quando esplicitamente richiesto. Tale politica di scheduling dei nodi può essere ottenuta definendo sui nodi transienti una `taint` con effetto `NoSchedule` (vedi e applicando delle `tolerations` alla detta `taint` generale su tutti quanti quei pod la cui esecuzione voglia essere affidata ai worker nodi transienti.

```
kubectl taint nodes node1 transient=true:NoSchedule
```

Listing 5.2: *Esempio di* `taint` *con effetto* `NoSchedule` *su un nodo transiente*

```
- tolerations:
    - key: "transient"
      operator: "Equal"
      value: "true"
      effect: "NoSchedule"
```

Listing 5.3: *Esempio di* `toleration` *a* taint `transient=true`

Storage. I deployment di Kubernetes su IaaS hanno a disposizione servizi di storage direttamente forniti dall'infrastruttura cloud che li ospita. Questa è sovente ben integrata con k8s, così da offrire funzionalità di *auto-provisioning* e *mount* di volumi, tutte caratteristiche assenti nei nodi HPC. Come alternativa di persistenza, i pod possono essere configurati per accedere direttamente allo storage locale disponibile sul nodo, attraverso la specifica di volume con `hostPath`. In aggiunta, se è disponibile local storage, il servizio `local-static-provisioner` (disponibile in https://github.com/kubernetes-sigs) costituisce una valida alternativa.

6 Flussi di generazione dati TDM

In questo capitolo sono descritti alcuni dei flussi di acquisizione e generazione di dati implementati sulla piattaforma TDM. In particolare, vengono raccolti dati da sensori esterni – e.g., sensori puntuali e radar meteorologici – e quelli generati da simulazioni meteorologiche.

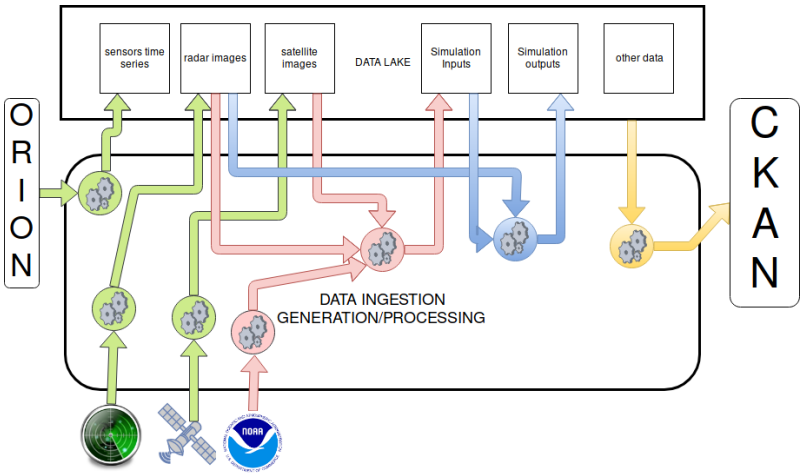

Figura 6.1: Flussi di generazione dati TDM.

6.1 Raccolta dati

6.1.1 Dati da sensori

La raccolta dei dati provenienti da sensori distribuiti avviene attraverso l'intermediazione dell'Edge Gateway. Data l'eterogeneità dei formati di dato, protocolli di comunicazione e policy di trasmissione usati dai diversi sensori, sia quelli attualmente utilizzati sia quelli che verranno usati in futuro, in prossimità delle stazioni di misura, nella periferia o *Edge* della rete di sensori, è stato introdotto un dispositivo che traduce questa molteplicità di linguaggi di comunicazione in un modello unico di formato e protocollo per l'ingresso nella piattaforma di elaborazione. Il formato comune è rappresentato dal set di modelli di dato armonizzati della piattaforma FIWARE/NGSI, mentre il protocollo utilizzato per la trasmissione è il publish/-subscribe MQTT. I messaggi così ricevuti dal broker MQTT Mosquitto vengono recuperati dal componente IoTAgent che provvede alla validazione e alla gestione logica del dispositivo, aggiornando il Context Broker Orion con informazioni quali dati acquisiti, nuovi sensori rilevati e/o aggiornati. Il Context Broker Orion provvede quindi a notificare il Persistence

Connector Cygnus ogni qual volta un dispositivo cambia i valori che costituiscono il proprio contesto, come i parametri rileveati e altri attributi. Dato che Orion non mantiene uno storico dello stato dei dispositivi, Cygnus provvede, per ogni evento notificato, a salvare i dati ricevuti da Orion su un backend precedentemente impostato. Al fine di disaccoppiare la gestione dell'evento dalla consumazione del dato, Cygnus inserisce i messaggi ricevuti in Topic di Kafka. Kafka è un broker di code di messaggi per l'elaborazione scalabile in ambito Big Data. I messaggio depositati nei Topic di Kafka vengono letti ed elaborati dai vari processi che lavorano in real-time sui dati. L'elaborazione è implementata tramite Apache Flink, un framework di ultima generazione per il processing distribuito real time. Sono stati scritti una serie di job Flink che processano i messaggi NGSI trasformandoli e aggregandoli temporalmente (sia su base oraria che giornaliera). Infine di dati, grezzi e aggregati, vengono memorizzati sul Data Lake e su CKAN. CKAN è uno strumento open source appartenente all'ecosistema Fiware per la creazione di siti web in grado di gestire open data. Consente la ricerca e il browsing dei dati, nonché la loro preview tramite mappe, grafici e tabelle.

6.1.2 Radar meteorologici

6.1.2.1 Radar meteo cittadino

L'acquisizione dei dati dal Radar Meteorologico è gestito attraverso un workflow di *Airflow*. Quando il Radar Meteorologico è attivo, le immagini vengono generate con un rate di una al minuto e vengono salvate su un repository server presso l'Università di Cagliari. Un workflow di sincronizzazione avviato periodicamente da Airflow controlla le immagini già presenti sulla piattaforma di elaborazione di TDM, ossia sul filesystem distribuito HDFS, e quindi sul repository remoto verifica se sono state generate nuove immagini dal radar. In caso affermativo le nuove immagini sono quindi copiate da quel sistema sull'HDFS a disposizione dei job che generano le simulazioni e le animazioni degli eventi meteorologici osservati.

Il tipo di sensore corrispondente utilizzato è descritto così

```
meteoradar_type = {
    "code": "0fd67c67-c9be-45c6-9719-4c4eada4acac",
    "type": "meteoRadar",
    "name": "Radar cittadino",
    "brandName": "Envisens",
    "modelName": "SuperGauge",
    "manufacturerName": "Envinsens Technologies",
    "category": ["sensor"],
    "function": ["sensing"],
    "controlledProperty": ["VMI", "SRI"],
}
```

Listing 6.1: *Tipo dato sensore per il radar cittadino.*

mentre il sensore vero e proprio ha questa descrizione

```
meteoradar_type = {
    "code": "0fd67c67-c9be-45c6-9719-4c4eada4acac",
    "type": "meteoRadar",
    "name": "Radar cittadino",
    "brandName": "Envisens",
    "modelName": "SuperGauge",
    "manufacturerName": "Envinsens Technologies",
    "category": ["sensor"],
    "function": ["sensing"],
    "controlledProperty": ["VMI", "SRI"],
}
```

Listing 6.2: *Descrizione del sensore radar cittadino.*

6.1.2.2 Mosaico radar protezione civile

Radar-DPC è la piattaforma del Dipartimento di Protezione Civile (DPC) della Presidenza del Consiglio dei Ministri che consente la fruizione attraverso Open Access Web Services di prodotti generati dal Centro Funzionale Centrale (CFC) in qualità di struttura tecnica del DPC. Tali prodotti sono realizzati attraverso catene operative proprie del CFC e implementate sulla base di una consolidata attività di ricerca della comunità scientifica nazionale e internazionale, oltre che da esperienze di omologhe strutture regionali e/o centri di Competenza del DPC. Le suddette catene operative consentono la produzione e la rappresentazione di specifici prodotti sulla base di dati grezzi provenienti dalla Rete Radar Meteo Nazionale (RRMN) e dalla rete delle stazioni pluviometriche e termometriche oltre che da altri strumenti (es. dati satellitari, fulminazioni, etc).

I dati sono assemblati come un mosaico che integra tutti i radar della protezione civile Italiana.

```
dpc_meteoradar_mosaic_type = {
    "code": "0fd67c67-c9be-45c6-9719-4c4eada4ffff",
    "type": "meteoRadar",
    "name": "Mosaic of dpc meteo radars",
    "brandName": "DPC",
    "modelName": "dpc-radar-mosaic",
    "manufacturerName": "Dipartimento Protezione Civile",
    "category": ["sensor"],
    "function": ["sensing"],
    "controlledProperty": ["VMI", "SRI"],
}
```

Listing 6.3: *Tipo dato sensore per mosaico radar protezione civile.*

Il sensore specifico

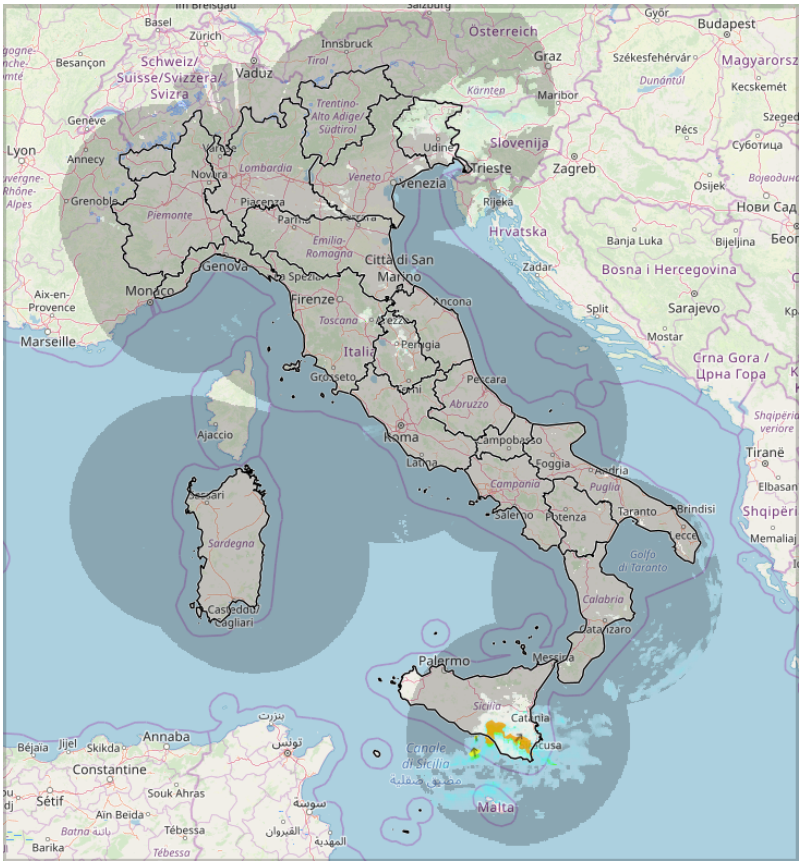

Figura 6.2: Mosaico precipitazione radar meteo della protezione civile.


```
{'code': '54d2c631-60ff-3f9a-879d-7b2a90b86184',
  'controlledProperty': ['VMI', 'SRI'],
  'geometry': {'coordinates': [[[4.537000517753033,
      47.856095810774605,
      0.0,
      1.0],
     [4.537000517753033, 35.07686201381699, 0.0, 1.0],
     [20.436762466677894, 35.07686201381699, 0.0, 1.0],
     [20.436762466677894, 47.856095810774605, 0.0, 1.0],
     [4.537000517753033, 47.856095810774605, 0.0, 1.0]]],
   'type': 'Polygon'},
  'geotiff_tags': {'GTModelTypeGeoKey': 2,
   'GTRasterTypeGeoKey': 1,
   'GeographicTypeGeoKey': 4326,
```

```
  'KeyDirectoryVersion': 1,
  'KeyRevision': 1,
  'KeyRevisionMinor': 2,
  'ModelTransformation': [[0.013249801624104052, 0.0, 0.0,
     4.537000517753033],
   [0.0, -0.009128024140684008, 0.0, 47.856095810774605],
   [0.0, 0.0, 0.0, 0.0],
   [0.0, 0.0, 0.0, 1.0]]},
 'grid': {'xsize': 1200, 'ysize': 1400},
 'name': 'dpc_meteoradar_mosaic',
 'nodecode': '0fd67ccc-c9be-45c6-9719-4c4eada4beaa',
 'stypecode': '0fd67c67-c9be-45c6-9719-4c4eada4ffff',
 'timebase': '2019-06-10T13:00:00Z',
 'timedelta': 300.0}
```

Listing 6.4: *Descrizione del sensore per il mosaico radar protezione civile.*

6.2 Simulazioni meteo

6.2.1 Modelli LAM, BOLAM e MOLOCH

BOLAM è un modello meteorologico idrostatico, che opera su un'area limitata del globo, sviluppato dal ISAC-CNR di Bologna. Le variabili prognostiche sono le componenti del vento, la temperatura, la pressione sulla superficie, l'umidità specifica e l'energia cinetica turbolenta. Il ciclo dell'acqua è descritto per mezzo di cinque ulteriori variabili prognostiche: le nubi di ghiaccio, quelle d'acqua, pioggia, neve e la grandine. Le variabili prognostiche definite nella griglia verticale hanno risoluzione maggiore nello strato limite atmosferico vicino alla superfice. La discretizzazione verticale è basata su un sistema ibrido di coordinate verticale, in cui le coordinate sigma terrain-following gradualmente tendono a una sistema di coordinate di pressione puro al crescere dell'altitudine. La discretizzazione orizzontale si basa su una griglia Arakawa-C sfalsata, in coordinate geografiche (latitudine-longitudine). Lo schema di avvezione attualmente implementato è WAF di Billet e Toro (1997). Il regime di diffusione orizzontale è del secondo ordine per tutte le variabili prognostiche, tranne la pressione superficiale. Le condizioni al contorno laterali sono applicate su un numero limitato (tipicamente 8) di righe di punti di griglia, utilizzando uno schema di rilassamento (Leheman, 1993) che assorbe efficacemente l'energia delle onde, contribuendo a ridurre la riflessione spuria dai confini laterali. BOLAM ha la capacità di effettuare simulazioni annidate, cioè con condizioni iniziale e al contorno ottenute mediante integrazione dello stesso modello (chiamato convenzionalmente "run padre") ad una risoluzione più bassa. La massima risoluzione è limitata dalla approssimazione idrostatica e dalla parametrizzazione della convezione, tipicamente circa 6-8 km. Per scale spaziali più piccole, viene utilizzato il modello MOLOCH che non utilizza l'approssimazione idrostatica. L'intero codice di BOLAM è scritto in Fortran 90. È stato parallelizzato, applicando la tecnica "splitting domain", ed è compatibile con ambienti di ela-

borazione in parallelo mpich2 e OpenMP. Il modello viene utilizzato operativamente da varie agenzie nazionali italianee servizi meteorologici regionali, ed è stato impiegato nelle previsioni in tempo reale per conto del Dipartimento Nazionale della Protezione Civile MOLOCH è stato sviluppato dall' ISAC-CNR di Bologna per fornire previsioni spazialmente dettagliate, rappresentando esplicitamente i fenomeni convettivi. Il modello MOLOCH integra il set completo delle equazioni per un'atmosfera non idrostatica e comprimibile utilizzando come variabili prognostiche la pressione, la temperatura, l'umidità specifica, le componenti della velocità verticale e orizzontale del vento e cinque specie di acqua condensata. La dinamica del modello è integrata nel tempo con uno schema implicito per la propagazione verticale delle onde sonore, mentre sono espliciti e time-split gli schemi per i restanti termini. L'avvezione è calcolata utilizzando lo stesso schema euleriano usato per BOLAM, di WAF di Billet e Toro (1997). La griglia verticale ha una spaziatura verticale che varia esponenzialmente con l'altezza. La microfisica è trattata sulla base della parametrizzazione proposta da Drofa e Malguzzi (2004). I processi fisici che determinano la tendenza nel tempo dell'umidità specifica, le nubi acqua/ghiaccio e il precipitato acqua/ghiaccio sono divisi in "veloci" e "lenti". I processi veloci coinvolgono trasformazioni tra umidità specifica e nube, mentre quelli lenti coinvolgono la produzione e la caduta di pioggia/neve/grandine. La temperatura è aggiornata imponendo l'esatta conservazione dell'entalpia a pressione costante. La precipitazione viene calcolata con lo schema stabile e dispersivo backward/upstream in cui la velocità è funzione della concentrazione.

6.2.2 Modello *Weather Research and Forecasting Model* (WRF)

Il modello ad aria limitata WRF è stato sviluppato da un partenariato collaborativo statunitense costituito dal National Center for Atmospheric Research (NCAR), National Oceanic and Atmospheric Administration(rappresentato dal National Centres for Environmental Prediction (NCEP) e il (poi) Forecast Systems Laboratory (FSL)), l'(allora) Air Force Weather Agency (AFWA), il Naval Research Laboratory, l'Università dell'Oklahoma e la Federal Aviation Administration (FAA). Tutte le informazioni relative al suo uso posso rivacarsi in http://www2.mmm.ucar.edu/wrf/users. Il modello è un sistema di previsione meteorologica numerica a mesoscala di nuova generazione progettato per la ricerca atmosferica e per le applicazioni di previsione operativa. È dotato di due core dinamici, un sistema di assimilazione dati e un'architettura software che supporta il calcolo parallelo. Il modello viene usato per una vasta gamma di applicazioni meteorologiche su scale da decine di metri a migliaia di chilometri. WRF offre nel settore delle previsioni operative una piattaforma flessibile ed efficiente dal punto di vista computazionale, avendo inglobato i recenti progressi in fisica, numerica e assimilazione dei dati forniti della comunità scientifica. WRF è attualmente in uso operativo presso l'NCEP e altri centri meteorologici nazionali, nonché in configurazioni di previsione in tempo reale presso laboratori, università e aziende private. WRF ha una grande comunità mondiale di utenti registrati (un totale cumulativo di oltre 39.000 in oltre 160 paesi. Il set di equazioni base del modello WRF, chiamato ARW (Advanced Reasearch WRF), descrive la fisica di un fluido completamente comprimibile, euleriano e non idrostatico con anche l'opzione idrostatica. È conservativo per le variabili scalari. Utilizza coordinate verticali di pressione idrostatica che seguono il terreno e che terminano nel condfine superiore con una superficie a

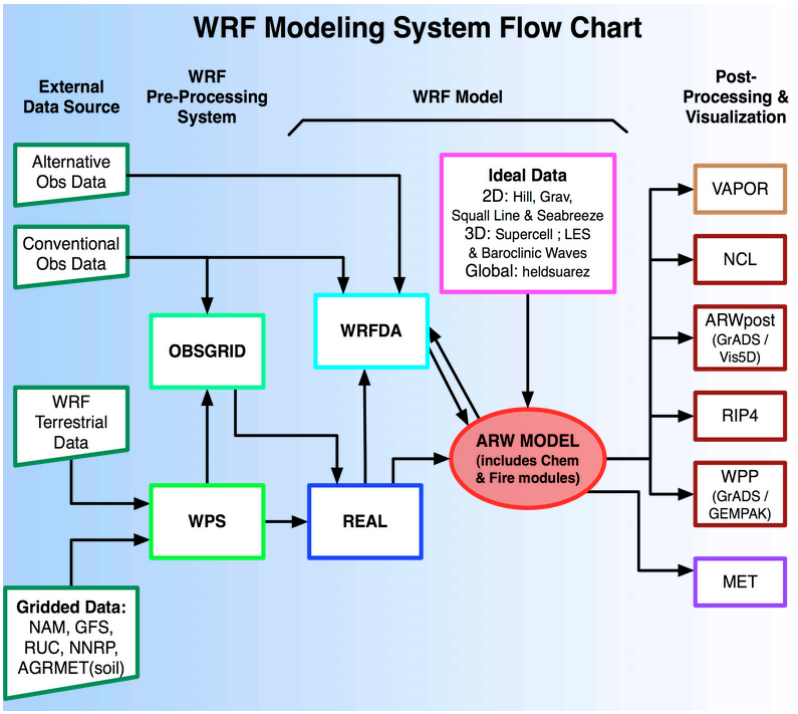

Figura 6.3: Schema della suite WRF. Sulla sinistra i dati di input che possono essere forniti. A WRFDA possono essere forniti dati radar (Aternative Obs Data) e osservazioni convenzionali. Il modulo obsgrid fornisce uno strumento per generare il formato supportato da WRFDA. Al modello WRF possono invece essere forniti le informazioni sul terreno e i dati provenienti da un modello globale. Queste informazioni vengono distribuite sulla griglia orizzontale dal modulo WPS ed infine distribuite sulla griglia verticale dal modulo REAL. IL modulo ARW MODEL è il solutore e può essere richiamato dal modulo di assimilazione WRFDA-4DVar. Sulla destra i vari strumenti per la visualizzazione

pressione costante. La griglia orizzontale è la griglia Arakawa-C. Lo schema di integrazione temporale utilizza lo schema Runge-Kutta di terzo ordine, e la discretizzazione spaziale impiega schemi di ordine dal 2^o al 6^o. Il modello supporta sia applicazioni idealizzate che reali con varie modalità per le condizioni al contorno. Supporta anche modalità di annidamento (nesting) unidirezionali (one way), bidirezionali (two ways) e con dominio a risoluzione variabile. Funziona su computer a singolo processore, condiviso e a memoria distribuita. WRF offre oltre al solutore dinamico una serie di componenti il cui scopo è quello di generare file di input (analisi) e condizioni al contorno per il modello. Questo sistema genera la condizione iniziale dei campi 3d e 2d del modello anche attraverso un aggiustamento del bilancio idrostatico. Il modello WRF supporta un set completo di opzioni per la fisica e il nudging dell'analisi e/o delle osservazioni. Il sistema di pre-processing ha il compito di definire la griglia WRF, a partire dall'analisi di dati reali e previsioni di un modello globale, interpolare

i dati di input sulla griglia e generare informazioni sull'elevazione e sul terreno nel dominio. I campi dipendenti dal tempo, ottenuti dell'integrazione del modello, sono costituiti da vento 3d, temperatura potenziale, vapore acqueo e un certo numero di campi 2d.

La WRF-3dVar e WRF-4dVar, cioè i moduli di assimilazione nello spazio e nello spazio/-tempo, possono essere utilizzati per assimilare le osservazioni nella condizione iniziale del modello. L'assimilazione dei dati è la tecnica con cui le osservazioni sono combinate con il prodotto di un modello meteorologico (la prima ipotesi o previsione di fondo) e le rispettive statistiche di errore per fornire una stima migliore (l'analisi) dello stato atmosferico. L'assimilazione variabile (Var) dei dati raggiunge questo obiettivo attraverso la minimizzazione iterativa di una funzione costo. Le differenze tra l'analisi e le osservazioni sono smorzate in base al loro errore. La differenza tra l'assimilazione di dati tridimensionali (3D-Var) e quadridimensionali (4D-Var) è l'uso di un modello numerico di previsione in quest'ultimo per l'evoluzione temporale. L' uscita standard dal modello WPS e WRF è in formato netCDF (uno dei formati WRF I/O). In figura 1 è schematizzato l' insieme degli strumenti forniti con WRF. Sulla sinistra i dati di input che possono essere forniti al modello. Ad WRFDA possono essere forniti dati radar (Aternative Obs Data) e osservazioni convenzionali. Il modulo obsgrid fornisce uno strumento per generare il formato supportato da WRFDA. Al modello WRF possono invece essere forniti le informazioni sul terreno e i dati provenienti da un modello globale. Queste informazioni vengono distribuite sulla griglia orizzontale dal modulo WPS ed infine distribuite sulla griglia verticale dal modulo REAL. IL modulo ARW MODEL è il solutore e può essere richiamato dal modulo di assimilazione WRFDA-4DVar. Sulla destra i vari strumenti per la visualizzazione

7 Misura delle Performance

In questo capitolo vengono riportati alcuni degli esperimenti effettuati per stabilire il livello di prestazioni del sistema.

7.1 Data lake

7.1.1 HDFS

In questa sezione, riportiamo alcuni risultati riguardanti le prestazioni del file system HDFS utilizzato. La configurazione del sistema è quella ottenuta utilizzando la chart ospitata nel repository `https://github.com/tdm-project/kubehdfs`, parametrizzato in modo da utilizzare nodi kubernetes specializzati (3 CPU, 50GB di RAM, 7.8 TB di disco). Questo porta alla seguente istanziazione:

- 1 namenode, definito come `StatefulSet`;
- 5 datanode, definiti come `DaemonSet`

Questa è configurazione di produzione di TDM.

L'esperimento è progettato come un gruppo di client che generano e scrivono dati sul sistema HDFS. I dati generati sono di tre differenti dimensioni: 10MB, 100MB, 1GB. Tutti i clienti lavorano in parallelo e asincronicamente generano ciascuno un flusso continuo di dati in scrittura sul file system HDFS.

Le misure fatte riguardano:

- banda di lettura/scrittura sui dischi collegati ai datanode HDFS;
- pressione di rete complessiva a cui viene sottoposto il sistema;
- il carico di CPU, rete a livello di singolo nodo datanode.

Le misure sono state ottenute ripetendo l'esperimento con numero di clienti pari a: 16, 32, 64, 128. Abbiamo `k8s-ge` per istanziare 10 nodi worker sul cluster Sun GridEngine.

Nell'immagine fig. 7.1 viene riportato il carico totale sulla rete. Come si può vedere, nell'esperimento siamo vicini a saturare la banda disponibile fra cluster HPC e OpenStack (20Gb/sec).

Nella figura 7.2 viene riportato l'IO su disco per ciascuno dei datanode HDFS. I diversi picchi corrispondono ai vari run eseguiti. La sequenza seguita è stata l'iterazione della sottosequenza rispettivamente di 16, 32, 64, 128 clienti. Si notino le operazioni di lettura/scrittura fra nodi, dovute alla replica dei blocchi del file system HDFS.

Nella figura 7.3 riportiamo, rispettivamente, il volume di dati trasmessi, ricevuti, il carico di CPU e di sistema di uno dei datanode del sistema HDFS. Tutti gli altri nodi del sistema mostrano lo stesso comportamento.

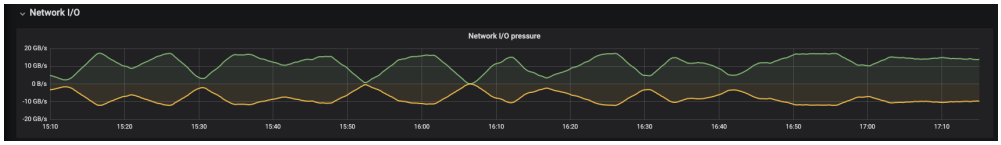

Figura 7.1: Pressione di rete durante i test HDFS. Le curve riportano il carico totale in lettura e scrittura durante i test. Come si può vedere si saturando la banda disponibile al link tra cluster OpenStack e HPC (20Gb/sec)

Come si può vedere, il sistema HDFS si comporta egregiamente utilizzando quasi completamente l'IO disponibile sui dischi. La misura aggregata è di circa 1.8GB/sec. Per altro, durante le misure, i carichi sui datanode si sono mantenuti tutti su livelli accettabili. Per altro, l'esecuzione dei test non ha portato particolare impatto sui datanode e sul sistema k8s complessivo.

7.1.2 Indicizzazione dei dati

Sono stati eseguiti una serie di esperimenti per validare la scelta di TimescaleDB (i.e., PostgreSQL con estensione per gestire le serie temporali) con PostGIS e dello schema implementato nel RDBMS. I test hanno messo alla prova anche le relativamente recenti funzionalità pertinenti aggiunte a PostgreSQL, come il tipo `jsonb` per gestire in maniera efficiente e funzionale i dati in formato JSON. Nei seguenti paragrafi sono descritti gli esperimenti e il loro esito.

7.1.2.1 Schema dei dati

È stato impostato lo schema illustrato nella Figura 7.4; da notare che il diagramma non rappresenta lo schema completo bensì solo la parte importante per rappresentare le serie temporali. Lo schema raffigurato è completo per quanto riguarda il modello relazionale dei dati ed è in grado di assicurare l'integrità dei dati catturati nel database.

La relazione `record` è configurata come *hypertable* in TimescaleDB. Ciò configura TimescaleDB per applicare delle particolari trasformazioni al modo in cui la relazione viene gestita all'interno della base di dati. Nello specifico, TimescaleDB automaticamente spezza la tabella in blocchi di misura relativamente contenuta e di cui sono tracciati i valori temporali massimi e minimi ivi registrati. L'algoritmo che ottimizza l'esecuzione delle query sfrutta questa disposizione dei dati su disco per esaminare solo i blocchi della tabella che sono pertinenti alla query. Per esempio, per eseguire una query che seleziona le lettura per un'ora specifica:

```
select time, source_id, value
    from record
    where time >= '2019-04-26 08:00:00' and time < '2019-04-26 09:00:00';
```

verrà esaminato solo il blocco della tabella che contiene quell'intervallo temporale, anziché l'intera tabella – e questo anche nell'assenza di indici. Grazie a questa strategia di TimescaleDB, solo due indici sono necessari in questo schema per eseguire in maniera efficiente le query richieste dal caso d'uso del progetto:

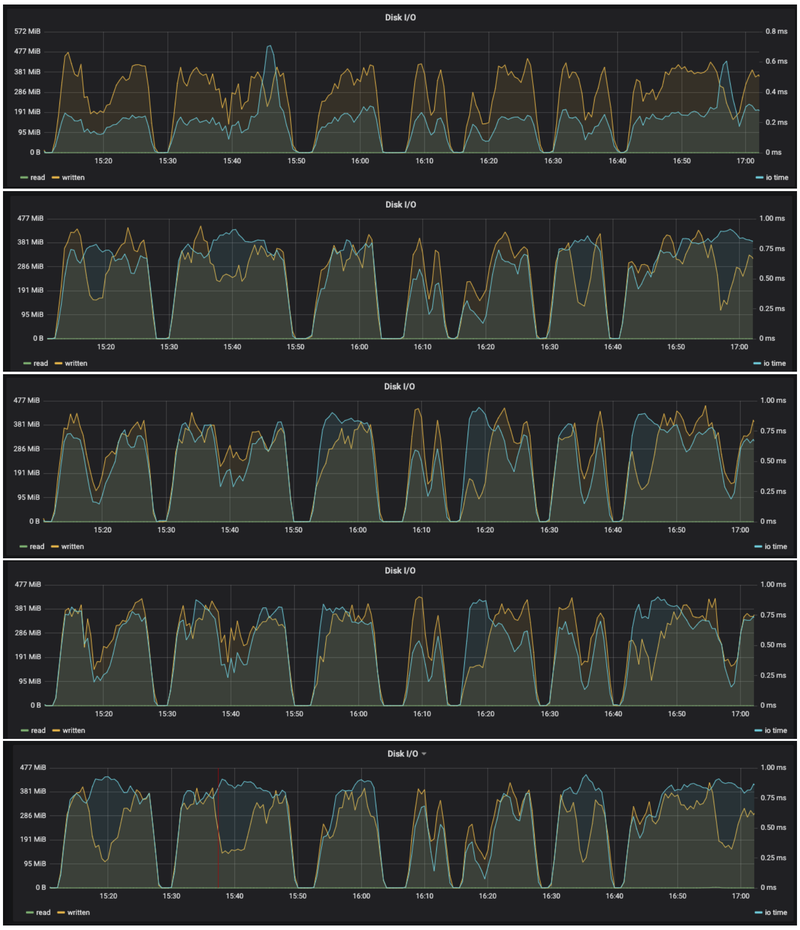

Figura 7.2: IO su disco per ciascuno dei datanode HDFS. I diversi picchi corrispondono ai vari run eseguiti. La sequenza seguita è stata l'iterazione della sottosequenza rispettivamente di 16, 32, 64, 128 clienti. Si notino le operazioni di lettura/scrittura fra nodi, dovute alla replica dei blocchi del file system HDFS.

```
CREATE INDEX ON record (source_id, time DESC);
ALTER TABLE source ADD PRIMARY KEY (sensor_id);
```

Il numero ridotto di indici migliora le prestazioni in inserimento, riduce le opportunità di contesa dei lock tra operazioni concorrenti, e riduce lo spazio complessivo occupato dalla base di dati.

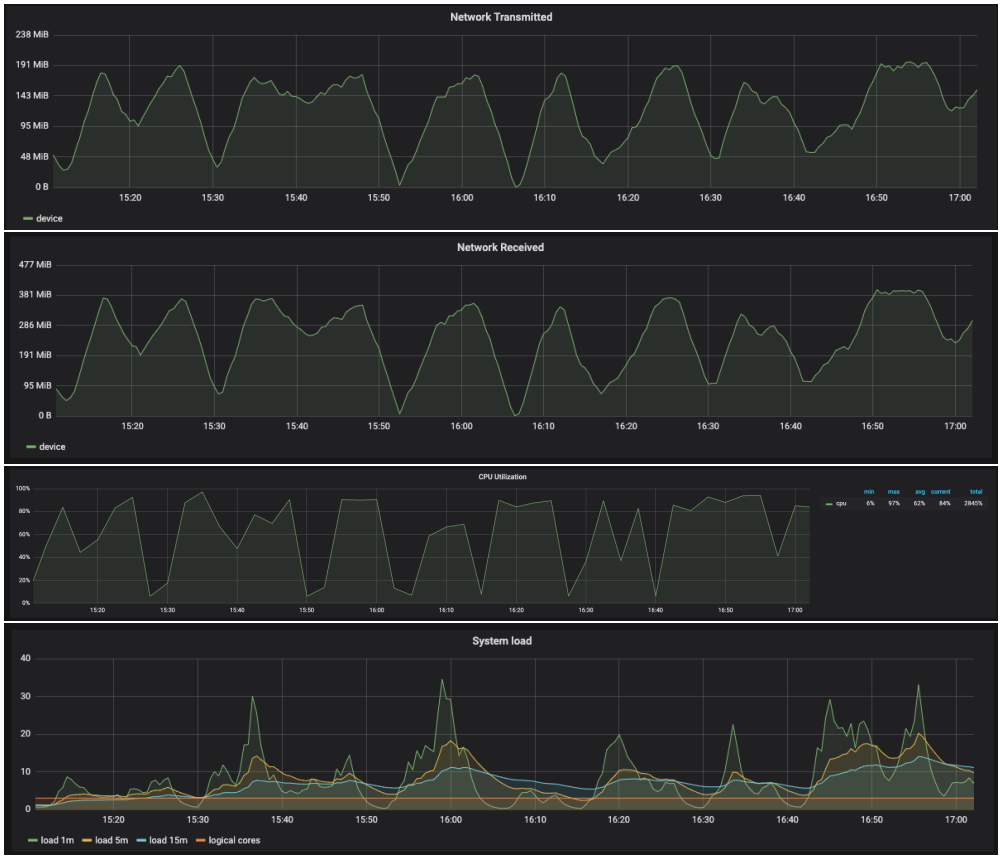

Figura 7.3: Volume di dati trasmessi, ricevuti, il carico di CPU e di sistema di uno dei datanode del sistema HDFS. Tutti gli altri nodi del sistema mostrano lo stesso comportamento.

7.1.2.2 Valutazione empirica

È stato disegnato un esperimento per misurare le prestazioni massime di registrazione di dati del database mentre simultaneamente multipli client simultaneamente lo interrogano con tutte le tipologie di query ipotizzate dai casi d'uso. Seguendo il modello standard che si sta seguendo per tutti i principali microservizi del sistema, il programma è stato implementato in Python e utilizza la libreria psycopg2[1] per collegarsi a PostgreSQL. Inoltre, il codice ha sfruttato le funzionalità più avanzate di psycopg2 per accelerare le query – e.g., `psycopg2.extra.execute_values` per accelerare la composizione delle query di inserimento, la conversione nativa del tipo JSON. Il programma legge un dataset da inserire da un file locale e lo inserisce più volte (per aumentare la mole di dati e rendere più significativo

[1] http://initd.org/psycopg

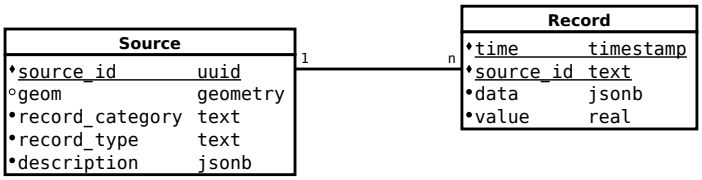

Figura 7.4: Particolare delle relazioni utilizzate per catturare le serie temporali dei sensori.

l'esperimento). Si sfrutta il multithreading di Python per evitare rallentamenti causati dalla lettura da file.

L'esperimento prevede:

1. creazione dello schema in un database vergine, inclusi gli indici;
2. avvio del thread di lettura da file per precaricare la coda di lavoro;
3. avvio dei thread di scrittura nella base di dati;
4. avvio dei thread di interrogazione;

Alla conclusione dell'inserimento di tutti i dati l'esperimento viene concluso con la chiusura di tutti i thread di interrogazione e il salvataggio dei risultati.

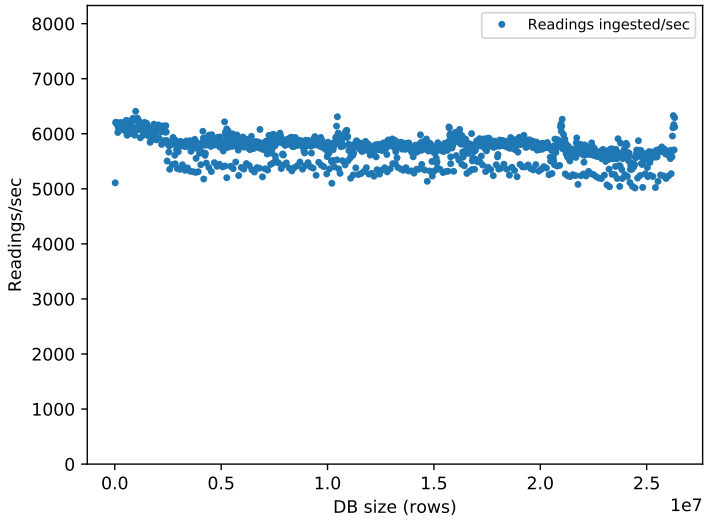

Figura 7.5: Numero di valori ingeriti dal database al secondo. La capacità di salvare e indicizzare i dati rimane stabile nonostante la crescita del database e nonostante l'utilizzo di risorse di calcolo non specializzate.

La figura 7.5 mostra il numero di letture di sensore inserite ogni secondo al crescere della collezione, da 0 a circa 27 milioni di valori. La velocità di inserimento rimane stabile e sostenuta attorno a 5500 valori al secondo per l'intera durata dell'esperimento, con un tasso sufficiente a supportare circa 33000 edge station TDM.

Simultaneamente al flusso di inserimento, sono state eseguite delle interrogazioni al database che simulano i casi d'uso previsti per la piattaforma. Il test ha effettuato un'interrogazione ogni secondo ciclando sulle seguenti (tra parentesi includiamo il nome della serie corrispondente nella figura 7.6):

1. Tutti i sensori registrati ("All sensor ids");
2. Tutti i sensori attivi in un'area geografica circolare in un specifico intervallo temporale di 24 ore ("Active sensors in geogr area");
3. La serie temporale relativa ad un sensore specifico e uno specifico intervallo temporale di 24 ore ("Data one sensor 1 day");
4. La serie di medie a finestre di 5 minuti (*time buckets*) della stessa serie temporale ("Data one sensor 1 day 5-min bucket avg").

La figura 7.6 mostra i risultati del test. I risultati mostrano che il tempo necessario per rispondere alle query rimane proporzionale alla quantità di dati da restituire, mentre è indipendente dalla quantità di dati nel database – suggerendo che il sistema sta scalando bene. I tempi di risposta rimangono sempre molto sotto il millisecondo.

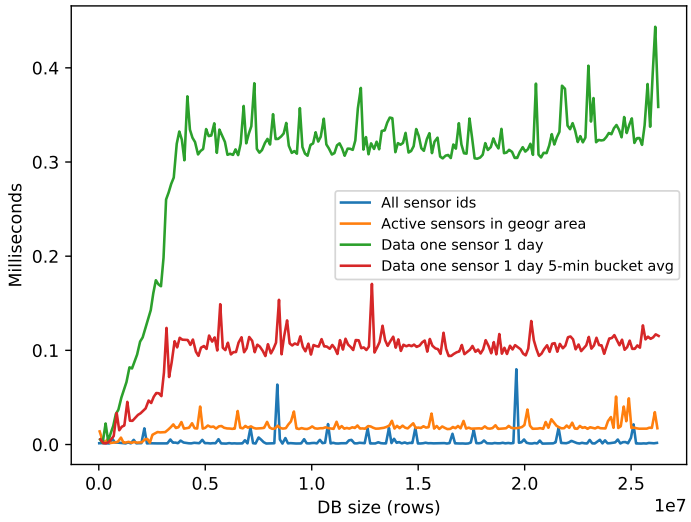

Figura 7.6: Tempo di esecuzione di vari tipi di query in concomitanza alla registrazione del flusso di dati in ingresso.

7.2 Simulazioni parallele a grande scala

7.2.1 Espansione dinamica del cluster Kubernetes su risorse HPC

Abbiamo eseguito una serie di esperimenti per misurare il tempo necessario per completare un operazione di espansione del cluster in funzione del numero di nodi da aggiungere. Pertanto, per ogni esecuzione, è stato lanciato un job Grid Engine che richiedeva un determinato numero di nodi. Abbiamo eseguito prove aggiungendo 1, 5, 10, 10, 15 e 20 nodi, misurando il tempo tra quando il job GE è stato messo in stato "Running"' e quando l'ultimo nodo ha raggiunto lo stato "Ready" su Kubernetes. Ogni prova è stata ripetuta 5 volte. Le figure 7.7 e 7.8 mostrano i risultati.

Le misurazioni indicano che i nuovi nodi sono generalmente pronti a funzionare in circa 40 secondi dopo che Grid Engine ha messo in esecuzione il job. Il numero di nuovi nodi richiesti non sembra avere un impatto sul tempo richiesto dall'operazione. Il fatto che `kubelet` a volte ha bisogno di essere riavviato per riuscire ad unirsi al cluster aumenta la varianza del tempo di completamento dell'espansione e crea alcuni nodi ritardatari. Inoltre, il tempo di attesa costante prima del comando di riavvio ha come effetto la distribuzione multimodale vista in fig. 7.8.

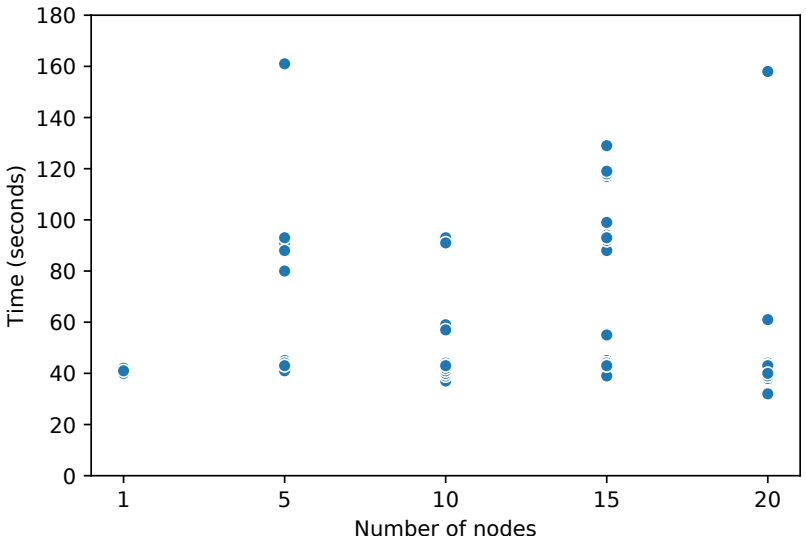

Figura 7.7: Scatterplot del tempo di espansione del cluster in funzione del numero di nodi.

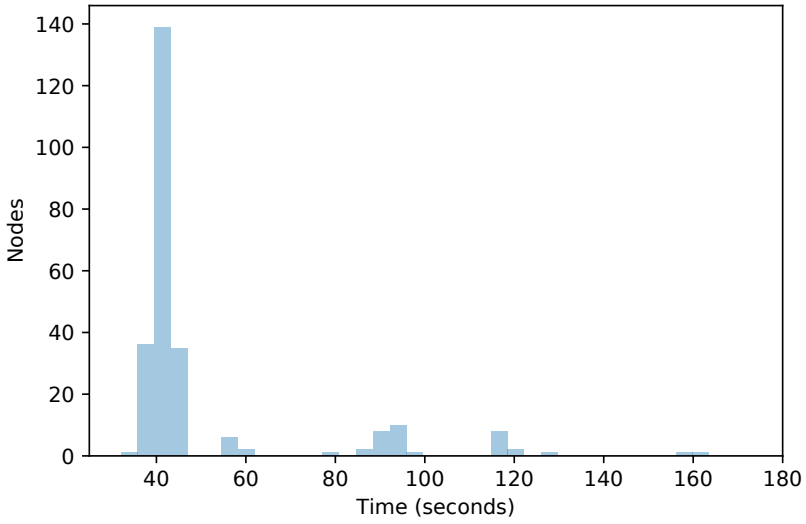

Figura 7.8: Istogramma dei tempi di annessione al cluster Kubernetes per i nodi Grid Engine. La distribuzione multimodale è dovuta al riavvio dei `kubelet` che non riescono a collegarsi dopo un tempo preimpostato di attesa.

7.2.2 Running

7.2.2.1 La catena operativa BOLAM

A partire dai dati forniti dal modello globale GFS (Global Forecast System) a 0.25° di risoluzione, abbiamo definito un esperimento giornaliero di previsione dello stato dell'atmosfera attraverso l'uso del modello meteo ad aria limitata BOLAM. La catena è costituita da uno strumento per lo scaricamento quotidiano dei dati del modello globale GFS e da un esperimento in cui è stato usato un dominio centrato sull'Italia di 578 x 418 punti con passo di 0.075° e 60 livelli verticali. Può essere usata l'analisi delle 00, 06, 12, 18 per l'inizializzazione del modello, mentre le condizioni al contorno vengono fornite ogni ora per un totale di 48h. Quindi la previsione può essere effettuata giornalmente alle 4 scadenze sinottiche.

7.2.2.2 La catena operativa MOLOCH

MOLOCH. A partire dalla previsione fornita dal modello ad area limitata BOLAM a 0.075° di risoluzione abbiamo definito un esperimento giornaliero di previsione dello stato dell'atmosfera attraverso l'uso del modello non idrostatico MOLOCH. Il dominio è centrato sull'Italia con 1156 x 1154 punti con passo di 0.0114°, 60 livelli verticali e 7 livelli nel suolo. L'analisi è basata sull'output di BOLAM a 3h dall'inizio della previsione principale (ossia quella effettuata con BOLAM) mentre le condizioni al contorno sono state fornite ogni ora per un totale di 24h.

Tabella 7.1: Parametrizzazione catena operativa BOLAM.

BOLAM	**Domain in space & time**
N° punti Latitudine	418
N° punti Longitudine	578
Estensione Latitudine	31°
Estensione Longitudine	43°
N° livelli verticali	60
N° livelli terreno	7
Time step	60s
N° time steps	2880
Tempo di previsione	48h

Tabella 7.2: Tempi di esecuzione catena operativa BOLAM.

N° di core	4 worker/nodo	8 worker/nodo	16 worker/nodo	24 worker/nodo
8	52.5’			
32	13.5’			
128	5.2’	5.4’	5.9’	5.7’

Tabella 7.3: Parametrizzazione catena operativa MOLOCH.

MOLOCH	Domain in space & time
N° punti Latitudine	1154
N° punti Longitudine	1154
Estensione Latitudine	13°
Estensione Longitudine	13°
N° livelli verticali	60
N° livelli terreno	7
Time step	25s
N° time steps	5760
Tempo di previsione	24h

Tabella 7.4: Tempi di esecuzione catena operativa MOLOCH.

N° di core	16 worker/nodo	24 worker/nodo
192	80.1’	
256	61.8’	
576	25.1’	23.75’
768		21.1’

7.2.2.3 La catena operativa WRF

A partire dai dati forniti dal modello globale GFS (Global Forecast System) a 0.25° di risoluzione abbiamo definito un esperimento giornaliero di previsione dello stato dell'atmosfera attraverso l'uso del modello meteo ad aria limitata WRF La catena è costituita da uno strumento per lo scaricamento quotidiano dei dati del modello globale GFS e da un esperimento in cui è stata usato un dominio centrato sull'Italia di 578x418 punti con passo di 0.075º e 50 livelli verticali. È stata usata l'analisi delle 00 per l'inizializzazione del modello, mentre le condizioni al contorno sono state fornite ogni ora per un totale di 48 h. I tempi di esecuzione sono in media di 5h su una macchina di un cluster linux.

Tabella 7.5: Parametrizzazione catena operativa WRF.

WRF	Domain in space & time
N° punti Latitudine	418
N° punti Longitudine	578
Estensione Latitudine	31°
Estensione Longitudine	43°
N° livelli verticali	32
N° livelli terreno	4
Time step	45s
N° time steps	3840
Tempo di previsione	48h

Tabella 7.6: Tempi di esecuzione catena operativa WRF.

N° di core	18 worker/nodo
18	92.4'

8 Conclusioni

Uno degli obiettivi principali del progetto TDM è di realizzare un'architettura scalabile per l'acquisizione, l'integrazione e l'analisi di dati provenienti da sorgenti eterogenee in grado di gestire i dati generati da un'area metropolitana estesa.

Ai due deliverable precedenti abbiamo descritto, rispettivamente, il nostro approccio all'acquisizione e aggregazione locale dei dati (D3.1) e alla distribuzione e presentazione dei dati (D3.2). Il presente rapporto ha completato la descrizione generale del sistema fornendo una descrizione dettagliata del design, delle componenti e delle performance del sottosistema di processamento dei dati.

Come discusso nei capitoli precedenti, tutti i risultati attesi di progetto sono stati raggiunti nei tempi e con le caratteristiche attese.

Trattandosi di un sistema in fase di continuo sviluppo e integrazione, le informazioni qui presentate potrebbero essere soggette a variazioni in futuro. Successivi aggiornamenti alla documentazione saranno messi a disposizione attraverso il portale del progetto http://www.tdm-project.it/ e il repository GitHub https://github.com/tdm-project/ .

Bibliografia

[1] V. Marmol, R. Jnagal, and T. Hockin, "Networking in containers and container clusters," in *Proceedings of netdev 0.1*, 2015.

[2] "Deploy a production ready kubernetes cluster," https://kubespray.io, April 2019, online; accessed 2019-Jun-21.

[3] M. E. Piras, M. del Rio, L. Pireddu, M. Gaggero, and G. Zanetti, "manage-cluster: simple utility to help deploy Kubernetes clusters with Terraform and KubeSpray." https://github.com/tdm-project/tdm-manage-cluster, April 2019, online; accessed 2019-Jun-21.

[4] M.E.Piras, L. Pireddu, G. Zanetti, M. Moro, "Container Orchestration on HPC Clusters," June 2019.

[5] A. Khalid, "hpc-wire: Bridging HPC and Cloud Native development with Kubernetes." https://www.hpcwire.com/solution_content/ibm/cross-industry/bridging-hpc-and-cloud-native-development-with-kubernetes/, April 2019, online; accessed 2019-Apr-26.

[6] W. Gentzsch, "Sun grid engine: Towards creating a compute power grid," in *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, ser. CCGRID '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 35–.

[7] *Sun N1 Grid Engine 6.1 Administration Guide*, https://docs.oracle.com/cd/E19957-01/820-0698/6ncdvjcmd, Oracle Inc., April 2019, online; accessed 2019-Apr-26.

[8] "Kubespray," https://github.com/tdm-project/kubespray/, April 2019, online; accessed 2019-Jun-21.