Agenzia nazionale per le nuove tecnologie, l'energia e lo sviluppo economico sostenibile

Ministero della Transizione Ecologica

Ricerca di Sistema elettrico

Architetture distribuite, interoperabili ed affidabili per la realizzazione della piattaforma Obserbot nell'ambito della LEC

Emiliano Casalicchio, Danilo Magliarisi

SAPIENZA UNIVERSITÀ DI ROMA

RdS/PTR(2021)/052

Architetture distribuite, interoperabili ed affidabili per la realizzazione della piattaforma Obserbot nell'ambito della LEC

Emiliano Casalicchio, Danilo Magliarisi

Dicembre 2021

Report Ricerca di Sistema Elettrico

Accordo di Programma Ministero della Transizione Ecologica - ENEA
Piano Triennale di Realizzazione 2019-2021 – III annualità
Obiettivo: Tecnologie
Progetto: Tecnologie per la penetrazione efficiente del vettore elettrico negli usi finali
Work package: Local Energy District
Linea di attività: 1.65 Architetture distribuite, interoperabili ed affidabili per la realizzazione della piattaforma Obserbot nell'ambito della LEC.
Responsabile del Progetto: Claudia Meloni, ENEA
Responsabile del Work package: Claudia Meloni, ENEA

Il presente documento descrive le attività di ricerca svolte all'interno dell'Accordo di collaborazione dal titolo "Architetture distribuite, interoperabili ed affidabili per la realizzazione della piattaforma Obserbot nell'ambito della LEC".

Responsabile scientifico ENEA: Alberto Tofani
Responsabile scientifico Sapienza Università di Roma: Emiliano Casalicchio

Indice

Sommario

L'analisi di grandi quantità di dati che sono generati nel Web è aumentato significativamente nell'ultimo decennio. In questo contesto, le piattaforme o reti sociali (social networks/platforms) forniscono uno strumento capace di acquisire dati da esse ed analizzarle. Facebook, Twitter, Instagram, Google, forniscono Application Programming Interfaces (APIs) che possono essere utilizzate dagli sviluppatori per collezionare ed analizzare dati. Per esempio, una emergenza sulla rete di distribuzione dell'acqua o dell'energia elettrica può essere identificata analizzando in tempo reale i tweet, come anche il sentimento degli utenti in tale circostanza.
Questo Progetto ha come obiettivo quello di migliorare Obserbot, un Sistema creato da ENEA per monitorare e raccogliere dati dalle reti sociali in modo automatico, ovvero senza necessità di alcun intervento umano, e per analizzarli. Obserbot è pensato come una piattaforma di supporto alle decisioni in caso di eventi critici (ad es. disastri ambientali, emergenze sociali o sanitari). Obserbot, se dotato di appropriati moduli di analisi, permette di predire o identificare eventi critici, e può anche essere utilizzato per determinare le azioni da intraprendere.
Un sistema come Obserbot deve avere delle proprietà non funzionali come scalabilità e resilienza, e proprietà funzionali come multitenancy o sviluppo ed integrazione continui (CD/CI). Per dotare Obserbot delle sopra elencate proprietà, in questo progetto ne viene rifattorizzata l'architettura software utilizzando il paradigma dei microservizi. La nuova versione di Obserbot, denominata Obserbot MS 2.0 verrà implementata, dispiegata e testata in un ambiente di produzione basato su Kubernetes, una piattaforma robusta e scalabile per l'orchestrazione container applicativi (e.g. Docker). Il prototipo implementato è stato testato in modo intensivo rispetto alle proprietà funzionali e non funzionali richieste.

1 Introduzione

Una delle sfide che ha dovuto affrontare la comunità scientifica nelle ultime decadi è stata quella di creare sistemi in grado di prevenire eventi potenzialmente disastrosi. Tali sistemi richiedono strumenti per la raccolta di informazioni che permettano di:

- acquisire lo stato delle infrastrutture, del territorio e delle persone coinvolte;
- valutare l'entità dei danni;
- indentificare le potenziali cause.

Le reti sociali (social networks) oggi sono una sorgente di informazioni estremamente utili agli scopi sopra elencati. I social network, infatti, sono un luogo dove gli utenti pubblicano testi e immagini in tempo reale, rendendole quindi un canale di informazione che può essere utilizzato per la gestione degli eventi. Può anche accadere che le autorità preposte alla gestione degli eventi critici vengano a conoscenza di tali casi attraverso i messaggi pubblicati sui social network piuttosto che dai canali ufficiali o convenzionali.

Una delle principali sfide nell'uso delle reti sociali a supporto della gestione delle emergenze è dato dall'enorme volume di messaggi, generati ad elevata frequenza, e che quindi devono essere acquisiti ed elaborati in modo automatico. Per risolvere questo problema, ENEA ha sviluppato uno strumento, chiamato Obserbot [1], per monitorare l'impatto di eventi indesiderati a partire dalla percezione degli utenti o cittadini.

Considerato il grande volume di dati prodotti dalle reti sociali, e l'elevata frequenza con cui essi vengono pubblicati, vi è la necessità di migliorare Obserboot rendendolo un sistema scalabile, resiliente e multi-tenant (ovvero fruibile contemporaneamente da più gruppi di utenti con diverse esigenze di analisi in diversi domini). Inoltre, è desiderabile che un sistema come Obserbot sia realizzato utilizzando un'architettura sofware che faciliti lo sviluppo e l'integrazione continua (Continuous Development and Continous Integration - CD/CI). Infine, Obserbot dovrebbe esporre interfacce pubbliche per l'integrazione con multiple sorgenti di dati e sstrumenti di analisi [1].

La prima versione di Obserbot è stata progettata e realizzata utilizzando un architettura software monolitica, che quindi non permette di soddisfare i requisiti appena menzionati (scalabilità, resilienza, multitenancy e CD/CI). **L'obiettivo di questo progetto è progettare, implementare e validare una nuova versione di Obserbot, denominata Obserbot MS 2.0, utilizzando lo stato dell'arte delle architetture software. In particolare, verrà re-ingegnerizzato Obserbot mediante il paradigma dei microservizi (microservices).** La scelta di questo paradigma software è un prerequisito per costruire un sistema che sia scalabile, multi-tenant e che supporti CD/CI. Inoltre, i microservizi sono tipicamente dispiegati usando i container applicativi (ad es. Docker) ed orchestrati mediante piattaforme come Kubernetes, che mettono a disposizione meccanismi per garantire la resilienza delle applicazioni. La soluzione proposta verrà sottoposta a test funzionali e non-funzionali (performance e scalabilita'). La validazione dovrebbe essere preferibilmente effettuata su di un caso di studio reale. Ci si riserva comunque la possibilità di validare il sistema realizzato mediante dati storici.

1.1 Organizzazione del Progetto e delle Attività

L'implementazione del progetto è organizzata nelle seguenti fasi:

- Fase 1: Selezione delle tecnologie allo stato dell'arte;
- Fase 2: Progettazione di Obserbot MS 2.0;
- Fase 3: Implementazione di Obserbot MS 2.0;
- Fase 4: Test funzionale e prima validazione;
- Fase 5: Deployment nell'ambiente di produzione;
- Fase 6: Test di performance e scalabilità, validazione finale.

Le Fasi 3 e 4 verranno iterate due volte. Nella prima iterazione verrà prodotta una prima versione di Obserbot MS 2.0, verrà quindi testata. Nella seconda iterazione verranno implementate eventuali azioni correttive, verrà eseguito un secondo test funzionale ed una prima validazione. Inoltre, le Fasi 3 e 4 faranno uso di un sistema di sviluppo, meno complesso e performante dell'ambiente di produzione, che verrà invece utilizzato nelle Fasi 5 e 6.

Nelle sezioni che seguono diamo una descrizione concisa delle Fasi 1 – 6. Per i dettagli è possibile consultare [13].

1.2 Analisi delle tecnologie allo stato dell'arte

La tecnologia principale utilizzata in questo progetto sono i microservizi. I microservizi sono uno stile architetturale software che struttura un'applicazione come un insieme di componenti software che hanno le seguenti caratteristiche:

- hanno elevata manutenibilità e testabilità
- sono debolmente accoppiati
- sono dispiegabili (deployable) in modo indipendente
- sono organizzati attorno alle capacità di business
- sono gestiti (progettati, implementati e manutenuti) da un gruppo di lavoro di piccole dimensioni

Inoltre, l'architettura a microservizi permette la distribuzione rapida, frequente e affidabile di applicazioni complesse e di grandi dimensioni, e consente ad un'organizzazione di evolvere il proprio stack tecnologico [2].

I microservizi, una volta progettati ed implementati vengono solitamente impacchettati in container applicativi per il successivo dispiegamento ed esecuzione. La tecnologia container utilizzata in questo progetto è Docker [8]. Google definisce il container come un pacchetto leggero che contiene il codice dell'applicazione, e le dipendenze, come versioni specifiche di runtime environment e librerie di linguaggi di programmazione necessari per eseguire il software. I container sono in grado di astrarre il software dall'ambiente in cui vengono utilizzati, creando così pacchetti eseguibili che possono essere utilizzati in qualsiasi piattaforma software. I container possono essere considerati delle macchine virtuali leggere, in cui il sistema operativo è installato all'interno del container stesso, così che il software sviluppato può essere distribuito ed eseguito su qualsiasi piattaforma, sia in Cloud, che on-premise. I vantaggi di questo approccio sono molti, ad esempio la condivisione delle risorse come CPU e RAM; la portabilità del software; l'isolamento di applicazioni. I container offrono migliori prestazioni e consumano meno risorse delle classiche macchine virtuali. Nei container, infatti, la virtualizzazione avviene a livello di processo, sfruttando le caratteristiche e le funzionalità del sistema operativo ospitante. Al contrario l'uso delle macchine virtuali richiede l'installazione di un substrato software, l'hypervisor. che filtra e traduce le richieste di accesso alle risorse generate dalle applicazioni e dal sistema operativo in esecuzione nelle macchine virtuali.

Un altro strumento essenziale per il progetto è il **Broker di Messaggi** (o semplicemente Broker nel seguito). Un Broker di Messaggi è un software che traduce e distribuisce i messaggi ricevuti dal mittente ad uno o più destinatari. All'interno del broker, alcune strutture dati consentono il corretto instradamento dei messaggi al destinatario. Esistono varie implementazioni di Message broker, ad esempio Apache Kafka [3], RabitMQ [4], Reddis [5] o Pulsar [6]. Kafka è uno dei broker più popolari. Scritto in Java e Scala, può acquisire flussi di dati da diverse sorgenti. Agisce come un sistema distribuito ad alte prestazioni, progettato principalmente per gestire facilmente, e in tempo reale, i dati in streaming. Dal confronto di Kafka con altri broker come RabbitMQ e Pulsar, è emerso che Kafka offre le migliori prestazioni in termini di throughput (messaggi al secondo processati) e di consumo delle risorse di sistema (Memoria e CPU) [7]. Il vantaggio di questo strumento è che fornisce un rigoroso ordinamento dei messaggi. Quando si verifica un evento, nel nostro

caso quando un nuovo tweet viene pubblicato, viene catturato istantaneamente da Kafka e memorizzato in una coda in base alla configurazione specificata. Indipendentemente dagli scenari di uso, Kafka rimane altamente scalabile, resiliente, tollerante ai guasti e sicuro. Per coordinare il funzionamento del message broker e dei microservizi che producono ed elaborano dati è necessario un sistema per la gestione delle informazioni (IMS) (e.g. configurazioni). L' IMS scelto è **Zookeeper** [9], che si integra facilmente con Kafka e Docker.

L'orchestrazione dei container verrà effettuata mediante **Kubernetes** [10]. Kubernetes si occupa della scalabilità, del failover e del dispiegamento delle applicazioni. Oltre a ciò, offre: gestione efficiente del carico di lavoro, spostamento e distribuzione del traffico su più container per garantire la stabilità del servizio; una gestione delle aree di memorizzazione (*storage*); una gestione dei container, che include la creazione, rimozione e riassegnazione delle risorse; una gestione del cluster di nodi, assegnando a ciascuno le risorse CPU e RAM necessarie; un sistema autorigenerante che garantisce il riavvio dei container che non rispondono alle richieste di controllo di disponibilità, deviando il traffico inoltrato a container non disponibili; una gestione delle informazioni sensibili tramite token OAuth e chiavi SSH, che possono essere distribuite e aggiornate senza riconfigurare l'intero cluster.

Per quanto riguarda **la memorizzazione dei dati collezionati** (i tweet) ed il risultato delle analisi sono stati impiegati due diversi tipi di database management system: MongoDB [11] e MySQL [12]. **MongoDB** è un database non relazionale orientato ai documenti (document-oriented). Permette di memorizzare degli oggetti senza bisogno di definire una struttura o uno schema del database per memorizzare documenti. I dati sono memorizzati come oggetti JSON (JavaScript Object Notation). MongoDB è un database distribuito che offre scalabilità ed elevata affidabilità. **MySQL** è un tradizionale database relazionale. Entrambi offrono elevate prestazioni in termini di numero di operazioni di lettura e scrittura al secondo (throughput).

Altre tecnologie utilizzate per la realizzazione di Obserbot 2.0 MS sono: **Flask**, un framework per creare REST API, fondamentali per l'accesso ai microservizi; **Kafka-python**, una libreria per programmare con Python in Kafka; **Tweepy**, una libreria Python per accedere alle API di Twitter; **Calico**, un container che offre strumenti per la gestione di rete in applicazioni distribuite.

1.3 Analisi di Obserbot

I component principali di Obserbot sono mostrati in Figura 1.
Il *Data Collector* raccoglie i dati dalle varie sorgenti (ad es. Twitter) e si interfaccia con il database MongoDB per memorizzare I dati raccolti, e con il broker Kafka che smista I messaggi in base alle esigenze di analisi. L'analisi dei dati viene svolta dal *Data Analyzer.* Il Data Analyzer si interfaccia con il database MySQL per memorizzare i risultati dell'analisi. Obserbot è stato progettato utilizzando un architettura software che non soddisfa le desiderate proprietà di scalabilità, elevata affidabilità e resilienza.

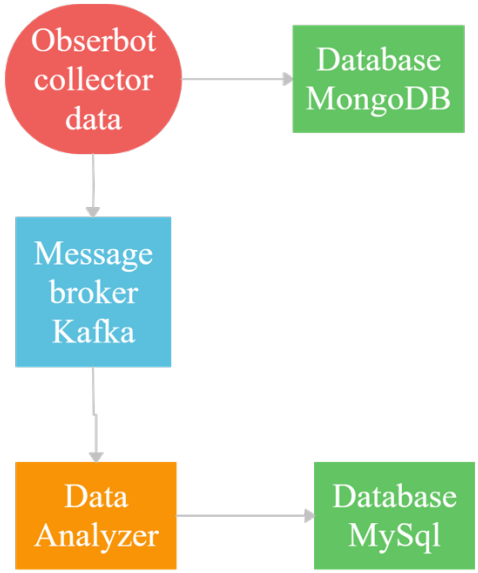

Figura 1. Componenti principali di Obserbot

1.4 Progettazione di Obserbot MS 2.0

1.4.1 Architettura software

I principali component di Obserbot MS 2.0 sono mostrati in Figura 2. Il *Data Collector*, che raccogli I dati dalle sorgenti desiderate, fa uso anche di un Database di parole chiave che permettono sia di generare le query appropriate per collezionare I dati (ad es. i tweet) sia per etichettare i messaggi in ingresso e permettere a Kafka di smistarli correttamente ai vari microservizi di analisi. Il Data collector invia i dati raccolti a Kafka e a MongoDB (query di scrittura). I *Data analyzer* sono microservizi che implementano le funzioni di analisi dei dati. L'uso di microservizi consente sia la scalabilità funzionale che orizzontale dei data analyzer. La prima permette la coesistenza di data analyzer che svolgono diversi tipi di analisi, la seconda consente di scalare, mediante replicazione, solo i microservizi di analisi che richiedono più risorse. Inoltre, la possibilità di avere piu' istanze dello stesso microservizio permette di aumentare l'affidabilità del sistema.

L'uso dei microservizi, di Docker e di Kubernetes, abilita anche la sostituzione "a caldo" dei Data analyzer e del Data Collector, ovvero è possibile sostituire un microservizio senza dover fermare momentaneamente Obserbot MS 2.0. Tale proprietà è estremamente utile nello svolgimento di operazioni di manutenzione evolutiva e di risoluzione di problemi (ad es. bug fix e patching di vulnerabilità).

L'architettura software di Obserbot MS 2.0 è mostrata in Figura 3. Il microservizio Producer implementa il Data Collector per interagire con Twitter (mediante le API standard di Twitter). E' possibile la coesistenza di diversi Data Collector per raccogliere dati da sorgenti multiple (ad es. altri social networks). A valle di Kafka abbiamo i microservizi Consumer, suddivisi in due gruppi. Nel Gruppo 1, abbiamo i Data Analyzer. Nel Gruppo 2 abbiamo i microservizi per la memorizzazione dei Tweet all'interno di un database (MySQL nella Figura). E' comunque stato progettato ed implementato anche un microservizio per memorizzare I dati in MongoDB.

Ai risultati delle analisi o ai dati grezzi si può accedere mediante un cruscotto Web e delle REST API progettate appositamente.

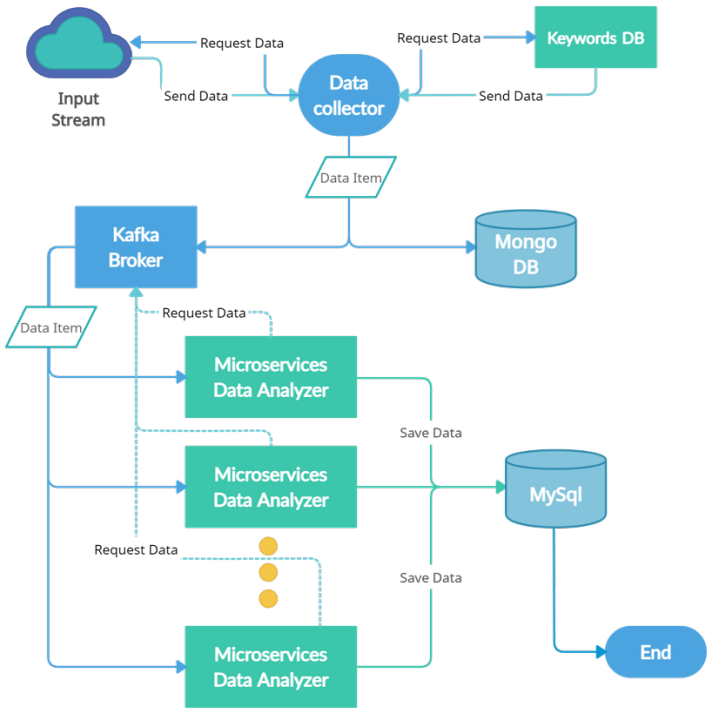

Figura 2. Principali componenti di Obserbot MS 2.0

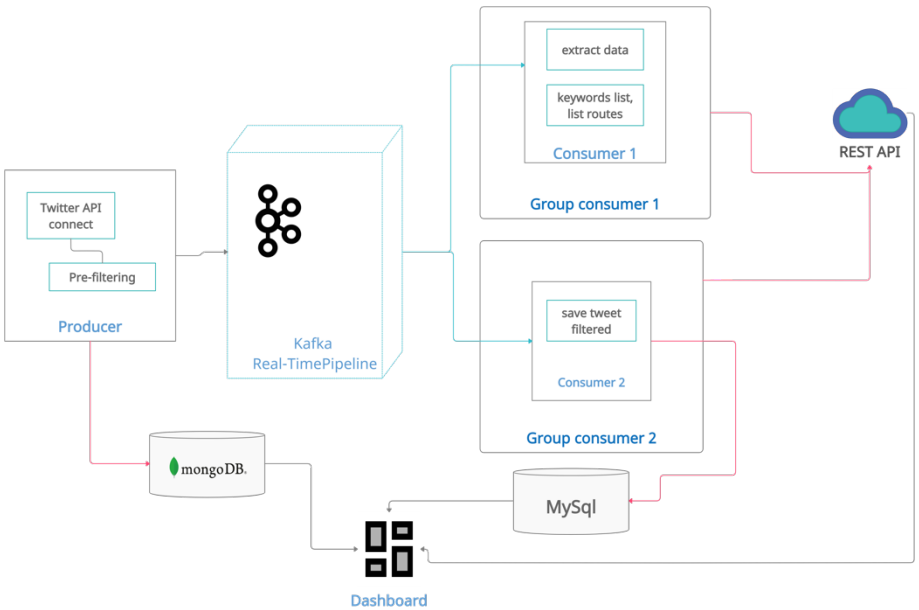

Figura 3 Architettura software di Obserbot MS 2.0

1.4.2 Microservizi

Lo schema completo dei microservizi realizzati è mostrato in Figura 4. Ogni microservizio rientra in un dominio che raggruppa tutti quelli di una determinata categoria. Sono state individuate quattro categorie principali che comprendono le fasi di *estrazione dei dati*, *salvataggio nel datastore*, *parole chiave* e *l'elenco di tutti i collegamenti ipertestuali* che fanno riferimento alle API REST a cui è possibile accedere. Complessivamente sono stati creati undici microservizi. Qualsiasi messaggio che entra nel sistema e passa

all'interno della coda Kafka viene serializzato nel formato JSON e quindi distribuito ai microservizi. Per ogni messaggio ricevuto da Twitter c'è quindi un file JSON che contiene dati organizzati in coppie chiave-valore. Per ogni valore, ottenuto in base alla chiave cercata, il microservizio estrae il suo contenuto per renderlo disponibile tramite l'API REST. Ne segue quindi, che per ogni chiamata effettuata alla specifica API, verrà generata una richiesta GET al microservizio che risponderà con il corrispondente valore cercato. In alcuni casi vengono utilizzate richieste POST che danno la possibilità di inserire dati da un servizio all'altro. Il framework Flask viene utilizzato per abilitare la comunicazione tra i servizi generati, e quindi rendere disponibili le API per ogni microservizio. Con questo strumento, abilitato su ogni componente, è sufficiente richiamare l'URL del servizio scelto ed effettuare una richiesta GET, POST, PUT o DELETE. Ogni richiesta effettuata tramite questo framework è di tipo RESTful. Vediamo nello specifico cosa contiene ogni dominio e quali compiti vengono eseguiti al suo interno.

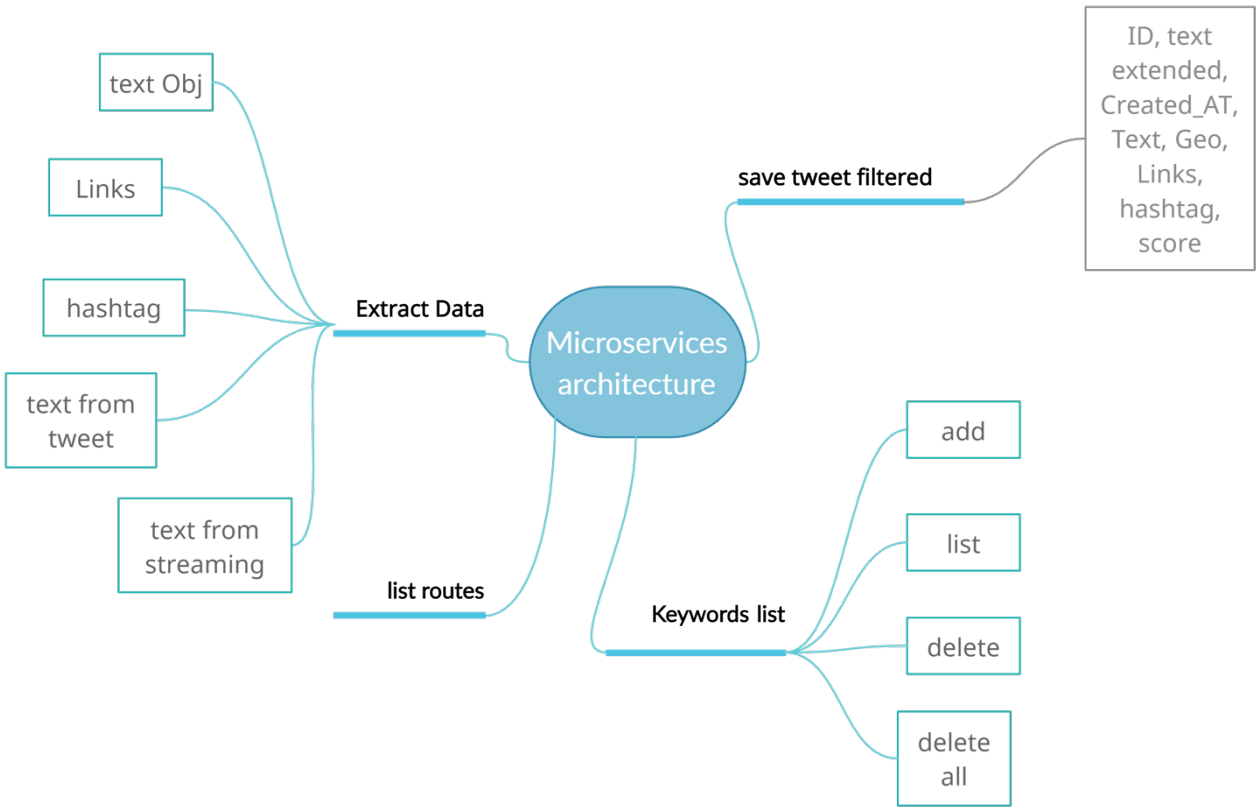

Figura 4 Microservizi in Obserbot MS 2.0

Il dominio *Extract Data* riunisce cinque servizi che elaboreranno i dati contenuti all'interno di ogni singolo file JSON ricevuto dalla coda Kafka. Di seguito i servizi contenuti in Extract Data: *hashtag*, restituisce un elenco di tutti gli hashtag contenuti in un singolo messaggio; *link*, restituisce l'elenco di tutti i link contenuti in un messaggio; *text obj*, restituisce il testo del tweet distinguendo se lo stato del messaggio è re-tweet, testo da tweet, testo da streaming.

Il dominio *List Route* non ha altri servizi al suo interno, il suo scopo è quello di rendere disponibili tutti gli indirizzi a cui è possibile accedere ad ogni richiesta GET. Ad esempio, fornirà l'URL e la porta per accedere al microservizio hashtag contenuto all'interno di Extract Data.

All'interno del dominio *Save Tweet Filtered* è presente un servizio che si occupa dell'interfacciamento con il database MySql. Nel DB viene salvato il messaggio ricevuto da Twitter ripulito da dati non utili al sistema Obserbot. All'interno della tabella Tweet di Mysql saranno presenti nove campi per ogni messaggio: *ID*, chiave

primaria che identifica univocamente il messaggio; *Created_AT*, la data di creazione del messaggio; *Text*, il testo dei tweet in forma ridotta; *text_Extended*, il testo dei tweet in forma estesa; *Geo*, il luogo da cui è stato generato il messaggio (questi dati non sono sempre presenti); *link*, i link contenuti nel messaggio; *hashtag*, gli hashtag contenuti nel messaggio; *punteggio*, il punteggio associato al messaggio in base alle parole chiave ricercate.

1.4.3 Il Cluster Obserbot MS 2.0

Affinché l'intero sistema soddisfi i requisiti di scalabilità, elevata affidabilità e resilienza, è necessario l'utilizzo di un cluster di macchine virtuali (VM). Tale cluster è stato implementato e gestito mediante la piattaforma OpenStack, già in uso in ENEA. Open Stack offre strumenti di orchestrazione delle macchine virtuali, quindi permette di creare un cluster di VM su cui poi dispiegare un cluster Kubernetes per l'orchestrazione dei container che implementano Obserbot MS 2.0. Questo doppio livello di virtualizzazione garantisce un ancor più elevata affidabilità e resilienza.

1.5 Implementazione di Obserbot MS 2.0

In questo capitolo riportiamo una sintetica descrizione dell'implementazione di Obserbot MS 2.0. Maggiori dettagli possono essere trovati in [13].

Come accennato nella sezione precedente Obserbot MS 2.0 viene eseguito su di un cluster di macchine virtuali gestito mediante OpenStack sul quale è stato dispiegato un cluster Kubernetes. La topologia di rete di base è quella mostrata in Figura 5.

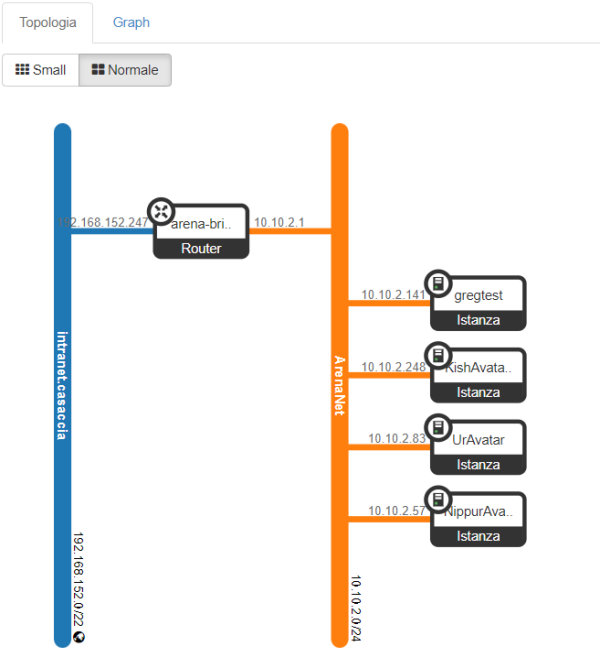

Figura 5 Cluster OpenStack

Per gli esperimenti in produzione sono state allocate quattro macchini virtuali. Su una di esse è stato installato il Kubernetes' Master Node (ovvero il control plan di Kubernetes) mentre le altre tre macchine virtuali svolgono il ruolo di Kubernetes' Nodes, ovvero eseguono Obserbot MS 2.0.

Il primo componente software dispiegato sul cluster Kubernetes è stato Kafka e a seguire Zookeper. Per il dispiegamento è stato utilizzato l'operatore *Strimzi* [14]. Kubernetes consente di eseguire più istanze di un software in esecuzione per aumentarne l'affidabilità e la resilienza. Seguendo questo principio, sono stati

configurati un cluster di tre nodi per Kafka ed un cluster di tre nodi per Zookeper. Rendendo così Obserbot MS 2.0 altamente affidabile e resiliente.

In Kafka, ai fini del testing e della validazione, è stata creata una sola topic, ovvero *tweets*. Come mostrato in Figura 6. Ovviamente, in caso di più sorgenti di dati, dovranno essere create più topic, una per ogni sorgente.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: tweets
  labels:
    strimzi.io/cluster: twitter-cluster
spec:
  partitions: 1
  replicas: 1
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
```

Figura 6 Creazione della topic Tweet in Strimzi

I microservizi, come precedentemente esposto, comunicano mediante REST API. Secondo i principi base dei servizi Restful, i metodi HTTP POST, GET, PUT, DELETE sono rispettivamente utilizzati per creare, leggere, aggiornare e rimuovere risorse (si veda Figura 7).

Action	HTTP Verb	Description
Create	POST	Create a new, unique thing
Read	GET	Read the information about a thing or collection of things
Update	PUT	Update the information about an existing thing
Delete	DELETE	Delete a thing

Figura 7 Metodi HTTP per servizi Restful

Ad esempio per richiedere la lista di keyword memorizzate nel Keyword DB è sufficiente invocare l'API keyword_list mediante una HTTP GET

```
http://KeywordDB_ADDR:5005/keywords_list
```

Dove KeywordDB_ADDR è l'indirizzo (IP o DNS) del database delle keyword. Invece, per creare una nuova keyword nel database delle keyword, è possibile invocare il metodo HTTP POST

```
requests.post('http://KeywordDB_ADDR:5005/keyword_list',
                    json = 'keyword' : 'new_keyword')
```

Per creare e gestire agevolmente le REST API per tutti i microservizi, è stato utilizzato il framework *Flask.*

I Database MongoDB e MySQL sono stati dispiegati con configurazione a singolo nodo, principalmente per le limitate risorse disponibili. Una configurazione in cluster per MongoDB e ad alta affidabilità per MySQL sono possibili e la loro implementazione è trasparente a quanto descritto in questo documento.
La creazione ed inserimento di un documento nel database MongoDB avviene secondo lo schema seguente (sintassi della libreria PyMongo):

```
db = client.TweetsDatabase
tweets_collection = db.tweets
tweet_id = tweets_collection.insert_one(tweet).inserted_id
```

Nella prima riga viene creato il database TweetsDatbase. Nella seconda riga viene creata la collection tweets nel database TweetsDatabase. Una collection può essere paragonata ad una tabella di un database relazionale (ad es. MySQL). Infine, nella terza riga viene inserito un tweet ricevuto da Kafka nel database.
Per quanto concerne il database MySQL, anche esso è costituito da una sola tabella che contiene tutte le informazioni di un tweet, separate in campi. Per l'installazione e la gestione di MySQL è stato utilizzato il framework Flask_mysqldb. Il codice che segue mostra lo schema per inserire un nuovo record (tweet) nel database MySQL.

```
mycursor = mysql.connection.cursor()
mycursor.execute("INSERT INTO tweets(ID, Created_AT, text,
          text_Estended, geo, links, hashtag, score)
          VALUES (ID, Created_AT, text, text_Estended, geo,
          links, hashtag, score) )
mysql.connection.commit()
mycursor.close()
```

I microservizi sono stati tutti containerizzati e despiegati in Pod Kubernetes. Il dispiegamento prevede la definizione di file che descrivono il servizio utilizzando il linguaggio YAML. In Figura 9 è mostrato un esempio di descrizione del servizio consumer. Si può notare che per default vengono dispiegate tre istanze del servizio e le richieste verranno distribuite tra esse (type:LoadBalancer). Inoltre, in caso di elevata frequenza di arrivo delle richieste, il componente Horizontal-Pod-Autoscaler (HPA) di Kubernetes permette di scalare automaticamente le funzionalità hotspot, ovvero HPA aggiunge o rimuove pod in funzione del livello di carico della CPU.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: obserbot-service-consumer
spec:
  selector:
    app: consumer
  ports:
  - protocol: "TCP"
    port: 6000
    targetPort: 5000
  type: LoadBalancer

---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: consumer-deployment
spec:
  selector:
    matchLabels:
      app: consumer
  replicas: 3
  template:
    metadata:
      labels:
        app: consumer
    spec:
      containers:
      - name: consumer
        image: consumer
        imagePullPolicy: Always
        ports:
        - containerPort: 5000
```

Figura 8 Definizione YAML del servizio consumer

1.6 Esperimenti

1.6.1 Test funzionale

Durante la fase di progettazione e sviluppo, prima del deployment sul cluster Kubernetes di produzione, è stato effettuato un test funzionale dei microservizi in occasione dell'evento "La Notte Rosa" tenutosi dal 26 Luglio al 2 Agosto 2021. Come si può evincere dalla Figura 10, questo evento ha generato un basso volume di tweet, che però ha permesso di verificare la correttezza delle funzionalità sviluppate.

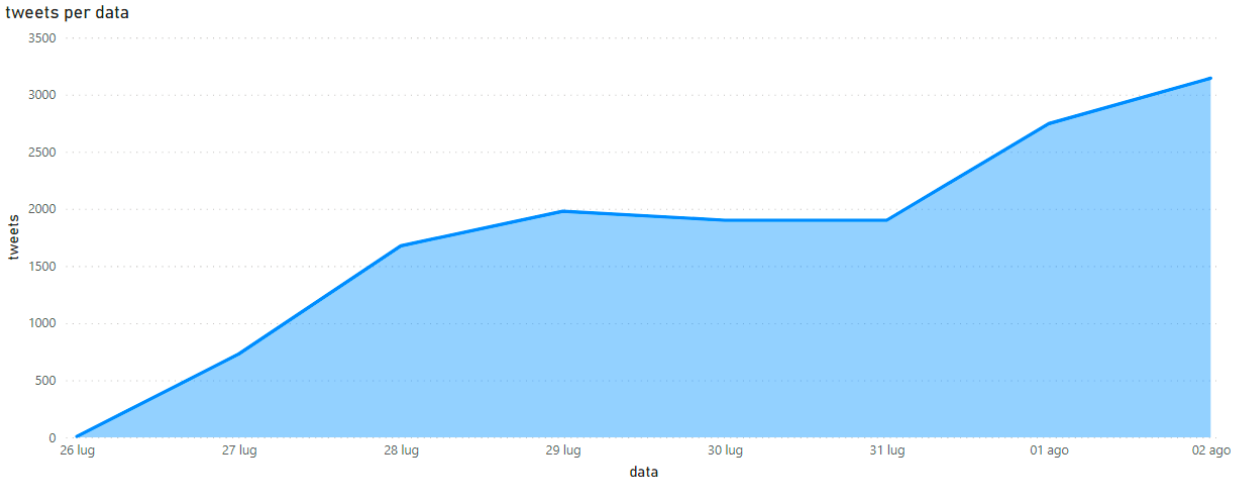

Figura 9 Andamento cumulativo del numero di tweet durante "La Notte Rosa"

1.6.2 Test di scalabilità e resilienza

Il secondo gruppo di esperimenti è stato concepito per verificare il corretto funzionamento e le proprietà di scalabilità e resilienza di Obserbot 2.0 MS nell'ambiente di produzione descritto nella Sezione 1.5.

Sono stati effettuati due test di scalabilità che differiscono per il tasso di arrivo dei tweet. Nello specifico: nel primo test è stato inviati al sistema un tweet al secondo; nel secondo test sono stati inviati due tweet al secondo. Questo basso tasso di arrivo (rispetto a situazioni reali) è stato scelto per via delle limitate risorse messe a disposizioni per il deployment di Obserbot MS 2.0. Per ogni test sono stati immessi nel sistema 50.000 tweet ed è stato misurata l'utilizzazione della CPU. Il Kubernetes HPA è stato configurato per aggiungere un nuovo Pod, ovvero una nuova istanza di microservizio, quando l'utilizzazione della CPU del pod raggiunge il 30%.
Nel primo esperimento nessun Pod (e quindi nessun microservizio) supera l'utilizzazione del 30% della CPU e quindi non viene generato nessun evento che aziona l'autoscaling. Nel secondo esperimento, invece il microservizio Filtered supera la soglia di autoscaling per quattro volte, come si evince dalla Figura 11. A tali violazioni corrispondono altrettante azioni di scaling, aggiungendo fino a tre pod. Nella Figura 12 si vede che nel 46% delle osservazioni sono stati istanziati due Pod e nel 12% dei casi sono stati istanziati tre Pod.

Il test finale è stato effettuato per verificare la resilienza di Obserbot 2.0 MS. In questo scenario, per ogni microservizio sono stati dispiegati tre Pod, ed è stato disabilitato l'HPA. Sono quindi state simulate delle failure nei Pod, in modo che i microservizi in essi eseguiti non fossero in grado di rispondere alle richieste. La distribuzione delle failure è stata casuale. Kubernetes, mediante il suo meccanismo di self-healing, ogni volta che ha rilevato un container nello stasto di failure ha generato un nuovo Pod. Non sono stati osservati downtime, ovvero Obserbot 2.0 MS è sempre risultato disponibile nonostate l'introduzione di failure. Questo grazie alla presenza delle tre repliche ed al meccanismo di self-healing.

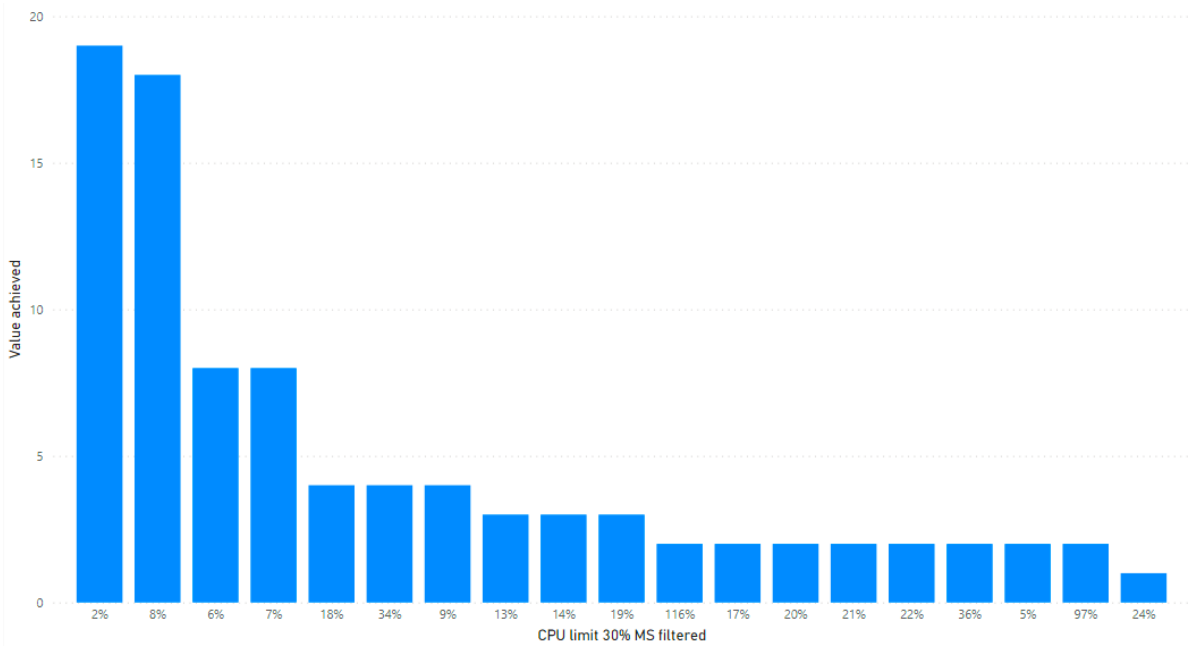

Figura 10 Utilizzazione della CPU osservata per il microservizio Filtered

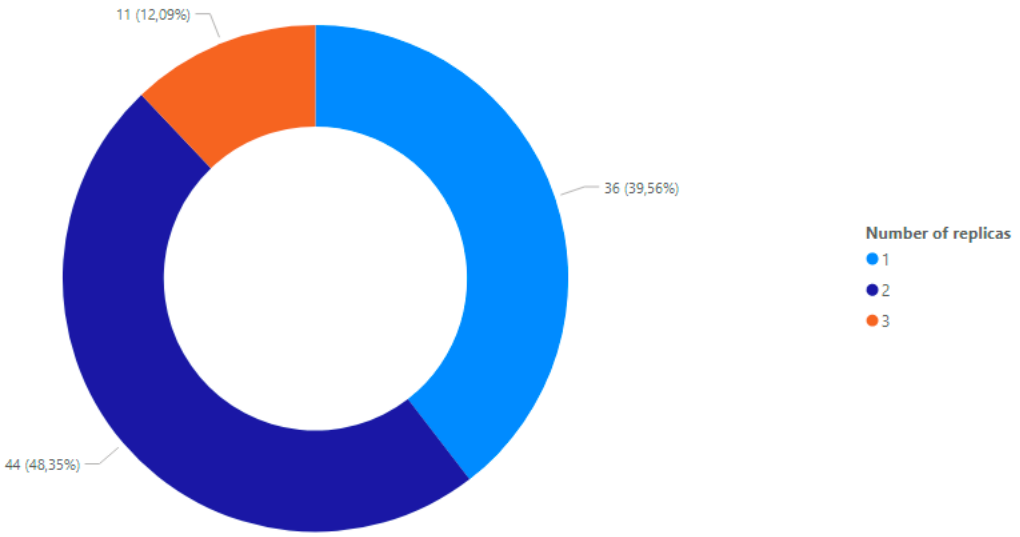

Figura 11 Distribuzione del numero di Pod allocati per il microservizio Filtered

2 Conclusioni

Questo progetto ha permesso di ottenere i seguenti risultati, confermati da evidenze sperimentali:

1) E' stato progettato un sistema modulare, scalabile, affidabile e resiliente in grado di collezionare e analizzare in tempo reale stream di dati, e di memorizzare tali dati in database relazionali e non relazionali.

2) La progettazione si è basata su di un'architettura a microservizi che, unita ad un deployment degli stessi mediante Docker container, orchestrati mediante Kubernetes, ha mostrato come la scalabilità funzionale e orizzontale permetta di allocando risorse addizionali solo per le funzioni hotspot, ovvero quelle che ricevono un maggior numero di richieste o che richiedono una maggior capacità computazionale rispetto alle altre.
3) L'uso dell'orchestratore Kubernetes ha anche mostrato come in presenza di failure, lo stato desiderato del sistema possa essere ripristinato automaticamente.

Per il futuro, si raccomanda di effettuare un test di scalabilità estensivo che:

- utilizzi un maggior numero di risorse computazionali (non disponibili durante il progetto);
- includa diverse sorgenti di dati, per contemplare il caso della concorrenza a livello di code Kafka e servizi producer;
- introduca un maggior numero di funzioni di analisi (e quindi relativi microservizi), per analizzare ulteriormente l'impatto del livello di concorrenza dei microservizi sulle prestazioni del sistema e validare il comportamento del meccanismo di autoscaling;
- consideri un'intensità del carico maggiore.

3 Riferimenti bibliografici

1. D'Agostino G., Tofani A., Di Martino B., Marulli F. (2021) Toward ECListener: An Unsurpervised Intelligent System to Monitor Energy Communities. In: Barolli L., Yim K., Enokido T. (eds) Complex, Intelligent and Software Intensive Systems. CISIS 2021. Lecture Notes in Networks and Systems, vol 278. Springer, Cham. https://doi.org/10.1007/978-3-030-79725-6_62.
2. Chris Richardson, Microservices Patterns, Manning Pub, October 2018, ISBN 9781617294549
3. Apache Karfka, kafka.apache.org (last access January 2022)
4. Redis, redis.io (last access January 2022)
5. RabbitMQ, rabbitmq.com (last access January 2022)
6. Pulsar, pulsar.apache.org (last access January 2022)
7. Kafka vs. Pulsar vs. RabbitMQ: Performance, Architecture, and Features Compared, https://www.confluent.io/kafka-vs-pulsar/ (last access January 2022).
8. Docker, docker.com (last access January 2022)
9. Zookeeper, https://zookeeper.apache.org/ (last access January 2022)
10. Kubernetes, https://kubernetes.io (last access January 2022)
11. MongoDB, https://www.mongodb.com (last access January 2022)
12. MySQL, https://www.mysql.com (last access January 2022)
13. Magliarisi D. (2022) Obserbot a microservices-based platform for collecting information from social media, Tesi di Laurea Magistrale in Computer Science, Sapienza Università di Roma.
14. STRIMZI, https://strimzi.io/, Kafka on Kubernetes in a few minutes (last access January 2022)

4 Abbreviazioni ed acronimi

API: Application Programming Interface

CD/CI: Continuous Development and Continous Integration

DB: Data Base

HPA: Horizontal Pod Autoscaler

IMS: Information Management System

JSON: JavaScript Object Notation

SSH: Secure Shell Protocol

REST: Representational state transfer

5 Biografia breve degli autori

Emiliano Casalicchio è professore associato presso il dipartimento di Informatica della Sapienza Università di Roma. Dal 2002 al 2014 ha svolto attività di ricerca presso l'Università di Roma Tor Vergata e la George Mason University (USA). Dal 2014 al 2017 è stato Associate Professor presso il Blekinge Institute of Technology, Svezia. E.Casalicchio svolge ricerca nell'abito di sistemi distribuiti su larga scala (Cloud Computing, Edge Computing, Fog Computing) studiando come garantire elevate prestazioni, scalabilità, resilienza e sicurezza. E.Casalicchio ha anche attivamente contribuito all'avanzamento dello stato dell'arte nell'ambito delle Infrastrutture Critiche. E.Casalicchio è autore di più di 120 pubblicazioni su riviste e in convegni internazionali, è revisore di articoli scientifici per riviste e conferenze internazionali, è revisore di progetti di ricerca a livello internazionale. Dal 2020 è presidente del Consiglio di Area Didattica in Informatica.

Danilo Magliarisi si è laureato in Computer Science (con lode) presso l'Università Sapienza di Roma. Ha maturato esperienza nella progettazione di sistemi distribuiti basati su container. Ha lavorato presso la RAI e presso il Dipartimento di Informatica dell'Università Sapienza di Roma presso cui è attualmente titolare di una borsa di studio post laurea per lo studio e la realizzazione di un sistema web per la pianificazione degli esami di profitto.