Sistemi Distribuiti e Cloud Computing:

Progetto A4: Microservices application vs. Serverless application

Valerio Crecco
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
valerio.crecco@students.uniroma2

Ludovico De Santis
Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata"
ludovico.desantis@students.uniroma2

I. Introduzione

Il progetto sviluppato consiste nella realizzazione di due differenti applicazioni, con funzionalità speculari, utilizzando, al fine di confrontarne le performance, due differenti approcci architetturali:

- microservizi;
- serverless;

In particolare, gli utenti della piattaforma possono iscriversi, al fine di poter effettuare la conversione di file. Nello specifico, è stata implementata una conversione dal formato csv al formato arff, al fine di migliorare ed ottimizzare le funzionalità di Machine Learning con tool specifici (e.g. Weka).

La conversione viene effettuata in maniera asincrona al fine di aumentare la flessibilità dell'applicazione, l'utente viene informato per mail dell'avvenuta conversione al fine di effettuare il download dalla piattaforma.

II. Architettura - Microservizi

L' applicazione, sviluppata in Python, è formata da 9 microservizi.

Per la realizzazione dei microservizi, realizzati tramite l'utilizzo di container, sono stati utilizzati Docker per la creazione di immagini e Kubernetes per la loro orchestrazione. Quest'ultima scelta è stata effettuata pensando alla flessibilità offerta dal tool, data la necessità di dover rendere l'applicazione scalabile e tollerante ai guasti.

Per la comunicazione di tipo asincrona è stato utilizzato il middleware message-oriented RabbitMQ, al fine di avere disaccoppiamento spaziale e temporale tra produttori e consumatori, oltre a permettere un disaccoppiamento di sincronia tra I consumatori e la coda di interesse. Per la memorizzazione di file è stato utilizzato MongoDB. Questa scelta è stata effettuata tenendo conto della grande flessibilità offerta da questo tool, ottimizzato per l'archiviazione di files, anche di grandi dimensioni, senza dover seguire uno schema relazionale, mentre per la memorizzazione degli utenti è stato utilizzato un database MySql.

III. Architettura - Serverless

L' applicazione, sviluppata in Python e JavaScript, è basata sull'utilizzo di diversi servizi offerti da AWS:

- AWS Lambda
- DynamoDB
- Amazon S3
- Amazon SQS
- Amazon API Gateway

Sono state realizzate 5 funzioni lambda, utilizzate in concerto con meccanismi ed automatismi forniti dalla piattaforma AWS, come ad esempio l'utilizzo di triggers ed eventi. Analogamente a quanto analizzato nei microservizi, la scelta di S3 per la memorizzazione di file è stata effettuata per motivazioni identiche alla scelta di MongoDB, mentre per la memorizzazione degli utenti è stato utilizzato DynamoDB.

IV. Microservizi – Funzionalità

- **Login**: permette di effettuare la registrazione e di conseguenza il login agli utenti per avere accesso alla piattaforma.
- **Upload** permette ad un utente registrato di effettuare l'upload di un file che si vuole convertire.
- **Converter**: permette di effettuare la conversione di file dal formato csv al formato arff, memorizzando il file convertito sul database MongoDB.
- **Notification**: permette di inviare una notifica via mail agli utenti che hanno effettuato un upload al fine di avvisare circa l'avvenuta conversione con successo del file stesso, indicando nel corpo della mail il fid del file per effettuare il download.
- **Download**: permette di effettuare il download in locale del file convertito.

V. Microservizi - Utilities

- **RabbitMQ**: permette di innescare I meccanismi di conversione di file e notifica di avvenuta elaborazione con successo attraverso l'utilizzo di due code distinte. Quando un file viene caricato, viene pubblicato un messaggio sulla coda di conversione (csv queue), così che I consumers possano provvedere ad effettuarla. Una volta ultimata, viene pubblicato un messaggio sulla coda di notifica (arff queue), a seguito del quale I consumers della coda stessa possono effettuare l'invio della mail di conferma all'utente che ha effettuato l'upload.

• **MySqlDB**: permette di effettuare la memorizzazione degli utenti e delle loro informazioni.

• **MongoDB**: permette la memorizzazione ed il retrieve dei file in maniera veloce ed affidabile, ottimizzando le performance.

VI. API Gateway

Offre le API per I microservizi direttamente accessibili dal client attraverso l'interfaccia web, la quale viene gestita da un server web Flask. In particolare, le API sono:

- /login
- /logout
- /register
- /upload_csv
- /download_arff

VII. Piattaforma Software

Docker – l'utilizzo di tale piattaforma ha consentito la virtualizzazione a livello di sistema operativo e di creare, testare e distribuire applicazioni con rapidità. L'utilizzo che ne è stato fatto è quello della creazione di immagini per I container da istanziare poi all'interno del cluster di Kubernetes.

Kubernetes – piattaforma utilizzata per l'orchestrazione e gestione di containers. Questa scelta è dettata dalla necessità di sfruttare tutte le peculiarità di questo framework, come ad esempio la caratteristica di self-healing (auto-placement, auto-restart, auto scaling). Offre una virtual network, ovvero ad ogni Pod è associato un indirizzo IP. Ciascun Pod può comunicare con gli altri proprio grazie a questo indirizzo IP.

I componenti utilizzati all'interno di K8s per lo sviluppo e gestione dell'applicazione sono:

- Deployment
- Service
- Statefulset
- Persistent Volumes

Un **Deployment** è un componente di Kubernetes che consente di descrivere il ciclo di vita dell'applicazione specificando, ad esempio, le immagini da utilizzare, il numero di pod necessari e relative modalità di aggiornamento.

Il **Service** permette di disaccoppiare il ciclo di vita di un Pod e il Service stesso. Questo significa che, anche se un Pod muore, il Service e l'indirizzo IP associati al Pod resteranno gli stessi. Definisce quindi come esporre dei Pod su una rete interna o esterna.

Gli **Statefulset** sono utilizzati per gestire applicazioni stateful, in particolare si occupano della gestione del Deployment e dello scaling di un insieme di pod, fornendo una garanzia di ordinamento e unicità di tali pod.

I **Persistent Volumes** sono una risorsa di Kubernetes che permettono di aggiungere uno strato di persistenza al cluster di kubernetes, poiché permettono di mantenere i dati indipendentemente dal ciclo di vita dei nodi facenti parte del cluster. Per l'utilizzo di Persistent Volumes è necessario un Persistent Volume Claim, ovvero una richiesta di storage da parte di un utente.

L'autoscaling di Kubernetes utilizzato in questa applicazione è l'Horizontal Pod Autoscaler HPA, il quale permette di scalare il numero di repliche dei Pod. L'algoritmo utilizzato si basa sull'osservazione dell'utilizzazione della CPU che, insieme all'utilizzo di memoria, costituiscono le uniche metriche nativamente supportate da Kubernetes. Il numero minimo di repliche impostato per ogni microservizio è di una, mentre il numero massimo è di cinque. L'unità di misura utilizzata da Kubernetes per la misurazione dell'utilizzo della CPU è il millicore (1000m = 1 Core), nel nostro caso la soglia di utilizzazione è posta a 200m (1/5 di 1 Core).

Minikube è stato usato per l'esecuzione di un cluster di Kubernetes in locale.

RabbitMQ è un middleware di comunicazione orientato ai messaggi che implementa il protocollo Advanced Message Queuing Protocol (AMQP). All'interno di tale applicazione questo middleware di comunicazione realizza un pattern di tipo competing consumers, in base al quale un messaggio viene consumato solo una volta ed esclusivamente da un solo consumer.
Sono state realizzate 2 code (persistenti):

• **csv-queue**: quando viene effettuato l'upload di un file, viene pubblicato un messaggio al fine di permettere ai consumers di effettuare la conversione.

• **arff-queue**: quando viene effettuata la conversione, viene pubblicato un messaggio per permettere di notificare l'utente dell'avvenuto completamento dell'elaborazione.

VIII. Serverless – Funzionalità

Le seguenti funzionalità sono state implementate tramite l'uso di AWS Lambda:

• **Login**: permette di effettuare la registrazione e di conseguenza il login agli utenti per avere accesso alla piattaforma.

• **Upload** : permette di effettuare l'upload dei file che si vogliono convertire, memorizzandoli in un apposito bucket di Amazon S3 (upload bucket).

• **Converter**: permette di effettuare la conversione di file dal formato csv al formato arff, memorizzando il file convertito su un apposito bucket di Amazon S3 (download bucket).

• **Notification**: permette di inviare una notifica via mail agli utenti che hanno effettuato un upload al fine di avvisare circa l'avvenuta conversione con successo del file stesso, indicando nel corpo della mail il nome del file per effettuare il download.

• **Download**: permette di effettuare il download in locale del file convertito.

IX. Serverless – Utilities

• **AmazonSQS**: permette di innescare I meccanismi di conversione di file e notifica di avvenuta elaborazione con

successo attraverso l'utilizzo di due code distinte. Quando un file viene caricato, viene pubblicato un messaggio sulla coda di conversione, così che I consumers possano provvedere ad effettuarla. Una volta ultimata, il file convertito viene caricato su un apposito bucket ed un messaggio viene pubblicato sulla coda di notifica, a seguito del quale I consumers della coda stessa possono effettuare l'invio della mail di conferma all'utente che ha effettuato l'upload.

• **DynamoDB** permette di effettuare la memorizzazione degli utenti e delle loro informazioni.

• **AmazonS3**: permette la memorizzazione ed il retrieve dei file in appositi buckets.

• **API Gateway**: permette l'accesso al sistema attraverso la realizzazione e distribuzione delle API Rest. In particolare:
  ◦ /login
  ◦ /register
  ◦ /verify
  ◦ /file-upload
  ◦ /file-download

X. Performance – Microservices vs. Serverless

Le applicazioni sviluppate sono state messe a confronto, tramite il tool di Load Testing **Locust**, al fine di analizzarne le performance. In particolare, sono state analizzate le prestazioni delle funzionalità di login, upload, download oltre ad una simulazione di un flusso di utilizzo completo del servizio, con l'obiettivo di testare il nostro sistema, non solo in scenari di singole funzionalità, ma anche in situazioni di utilizzo corale delle stesse. Gli scenari di utilizzo sono stati progettati considerando situazioni in cui vi fossero rispettivamente: 50, 100 e 150 utenti ad inviare richieste al sistema, in intervalli compresi tra 2 e 5 secondi.

Per quanto riguarda la funzionalità di Login, si può osservare come al variare degli utenti, le performance dell'applicazione serverless tendano a rimanere stabili, come si può osservare dai report ottenuti: Figura 1 (50 utenti), Figura 2 (100 utenti) e Figura 3 (150 utenti). Nel caso dell'applicazione a microservizi, invece, si ha un crescita dei tempi medi di risposta all'aumentare degli utenti (nel caso con 50 utenti – Figura 4, nel caso con 150 utenti – Figura 5).

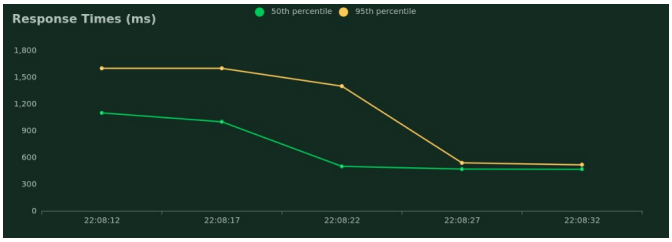

Figura 1

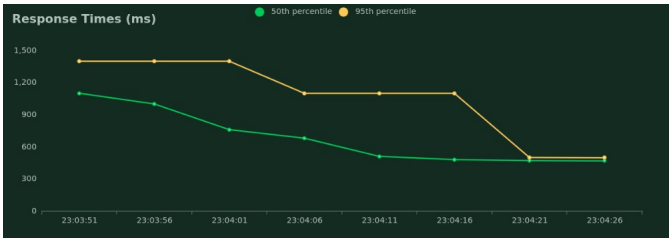

Figura 2

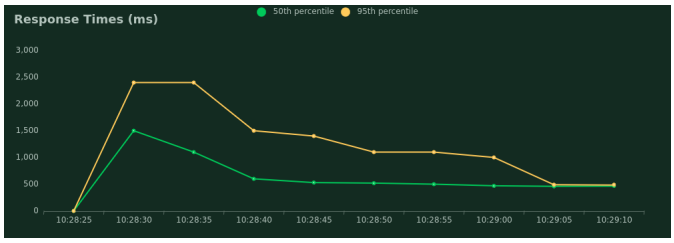

Figura 3

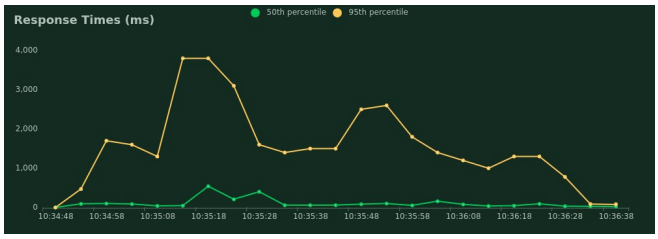

Figura 4

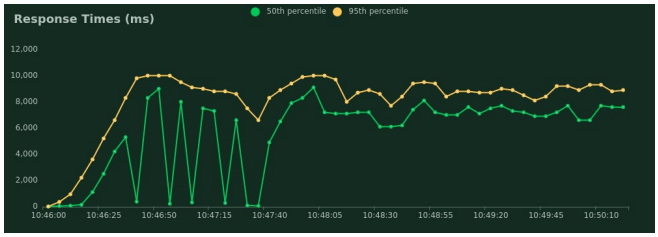

Figura 5

Per quanto riguarda la funzionalità di Upload, anche in questo caso, si può osservare come al variare degli utenti, le performance dell'applicazione serverless tendano a rimanere stabili, come si può osservare dai report ottenuti: Figura 6 (50 utenti), Figura 7 (100 utenti) e Figura 8 (150 utenti). Nel caso dell'applicazione a microservizi, invece, si ha un crescita dei tempi medi di risposta all'aumentare degli utenti (nel caso con 50 utenti – Figura 9, nel caso con 150 utenti – Figura 10).

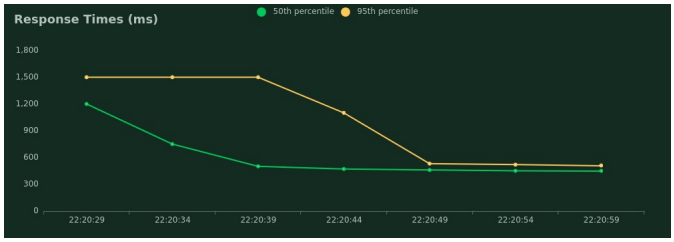

Figura 6

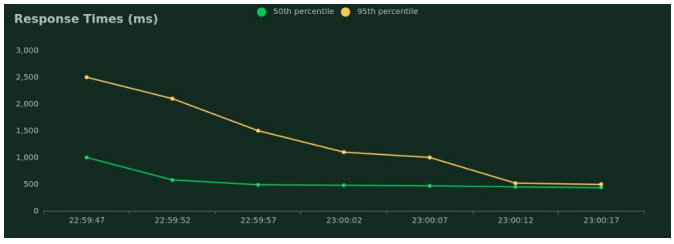

Figura 7

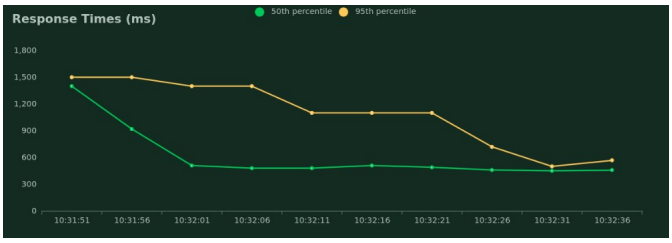

Figura 8

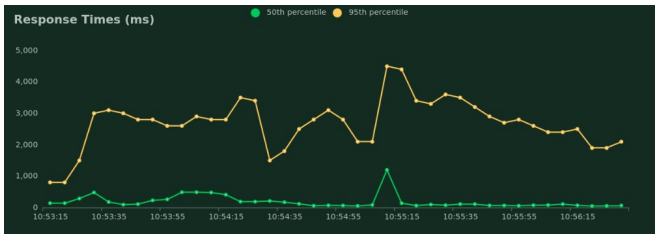

Figura 9

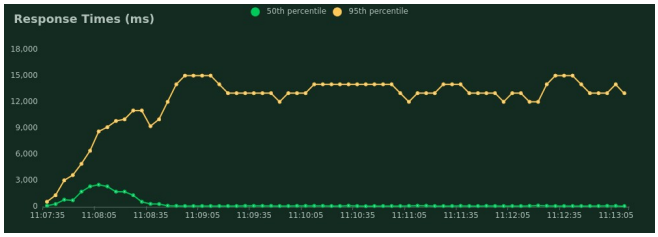

Figura 10

Per quanto riguarda la funzionalità di Download, anche in questo caso, si può osservare come al variare degli utenti, le performance dell'applicazione serverless tendano a rimanere stabili, come si può osservare dai report ottenuti: Figura 11 (50 utenti), Figura 12 (100 utenti) e Figura 13 (150 utenti). Nel caso dell'applicazione a microservizi, invece, si ha un crescita dei tempi medi di risposta all'aumentare degli utenti (nel caso con 50 utenti – Figura 14, nel caso con 150 utenti – Figura 15).

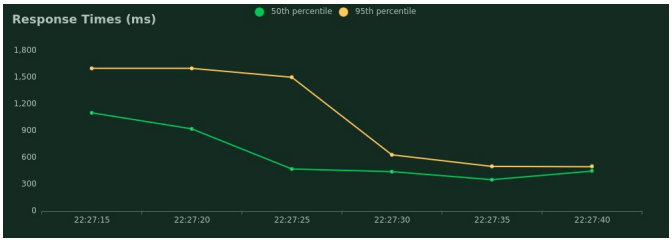

Figura 11

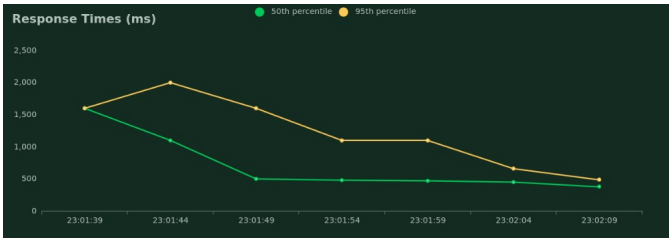

Figura 12

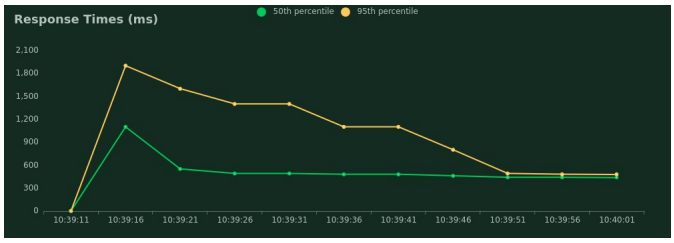

Figura 13

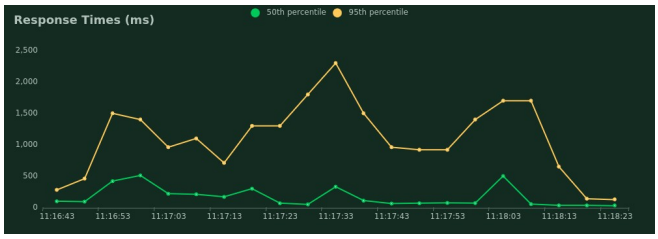

Figura 14

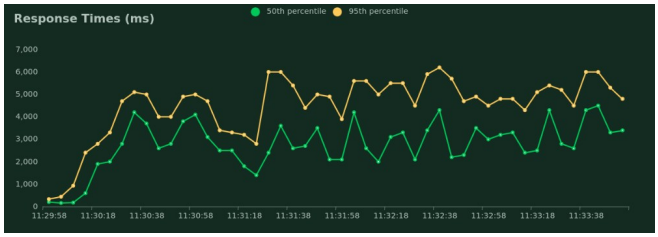

Figura 15

Infine, nello scenario di utilizzo corale di tutte le funzionalità, anche in questo caso, si può osservare come al variare degli utenti, le performance dell'applicazione serverless tendano a rimanere stabili, come si può osservare dai report ottenuti: Figura 16 (50 utenti), Figura 17 (100 utenti) e Figura 18 (150 utenti). Nel caso dell'applicazione a microservizi, invece, si ha un crescita dei tempi medi di risposta (nel caso con 50 utenti – Figura 19, nel caso con 150 utenti – Figura 20).

Figura 16

Figura 17

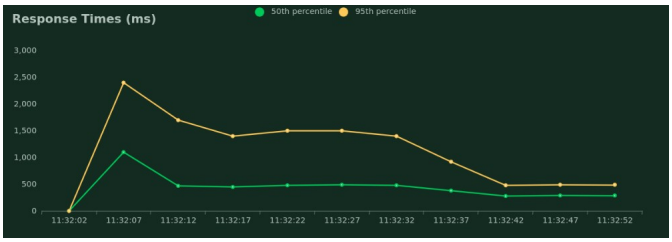

Figura 18

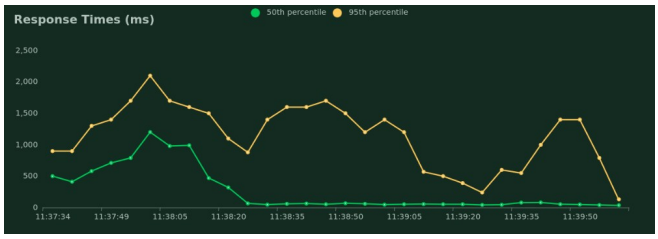

Figura 19

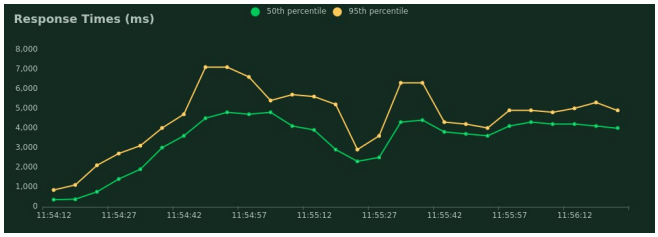

Figura 20

L’andamento generale dei tempi medi di risposta tende ad abbassarsi grazie all’uso di meccanismi di scaling. Nell’applicazione a microservizi viene utilizzato il Kubernetes HPA, un meccanismo di scaling orizzontale dei pods, che vengono allocati in maniera reattiva rispetto al monitoraggio dell’utilizzo delle virtual CPU allocate. Nel caso dell’applicazione serverless, invece, il meccanismo di scaling viene automatizzato da AWS in maniera proattiva. Si può quindi osservare che, nel primo scenario (microservizi) all’aumentare iniziale delle richieste, si osserva un aumento sostanziale dei tempi medi di risposta, dovuti alla saturazione delle risorse ed al fatto che il provisioning venga effetttuato solo a seguito della ricezione delle richieste e non a priori, ovvero non basandosi sull’utilizzo di modelli predittivi per stabilire il carico di lavoro del sistema. Nel secondo caso (serverless), invece, grazie all’uso di un approccio proattivo, si riescono ad avere dei tempi medi di risposta simili tra le varie simulazioni, legati al fatto che il sistema è sempre pronto a gestire differenti carichi di lavoro, allocando in modo preliminare, le risorse necessarie.

LIBRERIE

A. RABBITMQ – PIKA

È un’implementazione in Python per il protocollo AMQP utilizzato da RabbitMQ. Tale libreria, è stata utilizzata per l’intero sistema di comunicazione asincrona nell’applicazione, tra cui le azioni di publish e consume

B. GRPC - GRPCIO

È uno strumento Python che include il compilatore di protocol buffer ed un plug-in per la generazione dei codici server e client a partire dai file .proto. Tale libreria, è stata utilizzata per l’intero sistema di comunicazione sincrona nell'applicazione.

C. FLASK - FLASK

È un framework Python per lo sviluppo di applicazioni web. Tale libreria, è stata utilizzata in entrambi gli API gateway per interfacciare l’utente all’applicazione.

D. RESTAPI - REQUESTS

Mediante tale libreria, è stato possibile inviare richieste HTTP/1.1 ai server web Flask ed interagire con API REST.

E. MONGODB – GRIDFS - MONGOCLIENT

Librerie utilizzate per interagire con il database e per memorizzare files sul in maniera ottimizzata e veloce.

F. SMTPLIB

Libreria utilizzata per la generazione ed invio della mail di notifica di avvenuta conversione.