

# Cloud Computing - Report

## Drive on Google Cloud Platform

Andrea Bacciu (1747105), Valerio Neri (1754516),  
Riccardo Taiello (1914000)

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Progetto</b>	<b>3</b>
2.1	Frontend . . . . .	3
2.2	Java 11 - Spring . . . . .	3
2.3	Google Cloud Platform . . . . .	4
2.3.1	Google App Engine . . . . .	4
2.3.2	Cloud Storage . . . . .	4
2.3.3	Cloud Functions . . . . .	4
2.3.4	Firebase Auth & Accesso alle API e File . . . . .	5
<b>3</b>	<b>Accesso ai file</b>	<b>6</b>
<b>4</b>	<b>Flusso di informazione</b>	<b>7</b>
4.1	Mkdir, LS, Download, Upload . . . . .	8
4.2	Download ZIP . . . . .	9
4.3	Download di un file . . . . .	10
<b>5</b>	<b>Requirements</b>	<b>11</b>
5.1	R2 . . . . .	11
5.1.1	Scalability with respect to computation & high availability of the computation . . . . .	11
5.1.2	Scalability with respect to data & high availability of the data . . . . .	11
5.1.3	Security of data in transit . . . . .	11
<b>6</b>	<b>Esperimenti</b>	<b>12</b>
6.1	Test Setup . . . . .	12
6.2	Test . . . . .	13
6.2.1	App Engine . . . . .	13
6.2.2	Google Cloud Storage . . . . .	15
6.2.3	Cloud Function . . . . .	16
<b>7</b>	<b>Conclusioni</b>	<b>18</b>
7.1	Sezione differenze con la proposta . . . . .	18

# Capitolo 1

## Introduzione

L'obiettivo del progetto è quello di creare un sistema di file storage analogo a Google Drive/OneDrive/Dropbox, il quale, a seguito della login permette all'utente di gestire un proprio spazio personale e privato, salvato nel cloud.

Le funzionalità implementate sono le seguenti:

- Creazione cartelle
- Scaricare singoli file
- Scaricare una cartella
- Upload di file
- Cancellazione singoli file
- Cancellazione cartella (vuota o non-vuota)
- Preview del file nel caso di immagini

Per la realizzazione del progetto abbiamo deciso di utilizzare i servizi offerti da Google Cloud Platform (GCP), in particolare *Firebase Auth* è stato utilizzato per gestire l'autenticazione e la sessione, *Google Cloud Storage* è stato utilizzato per salvare e gestire file e cartelle, *Google App Engine* è stato utilizzato per far scalare il server web sviluppato in *Spring*, infine *Google Cloud Functions* sono utilizzate per la creazione dello zip nel caso del download di una cartella. I servizi sono stati scelti in modo da garantire Scalability e High availability, un'analisi accurata delle varie componenti del progetto è presente nel Capitolo [2](#).

Un video che mostra il frontend e le funzionalità messe a disposizione dell'utente è disponibile al seguente link <https://www.youtube.com/watch?v=6Uht4Emzqic>.

## Capitolo 2

# Progetto

In questo capitolo verranno descritte le varie componenti del progetto ed i servizi di *Google Cloud Platform* utilizzati.

### 2.1 Frontend

Il frontend è realizzato in Javascript, utilizzando JQuery e Bootstrap. Il frontend è composto da file statici, completamente indipendente dal backend, questo permette ai file del frontend di essere cached senza problemi.

Oltre a fornire all'utente l'interfaccia per poter interagire con l'applicativo e di conseguenza con i servizi di *Google Cloud Platform*, il frontend gestisce l'autenticazione con Firebase, ottenendo il session token che verrà passato al backend così che possa verificare che l'azione richiesta sia autorizzata.

### 2.2 Java 11 - Spring



Spring Boot è un framework Java molto utilizzato, il framework permette la costruzione di un backend con il minimo effort, utilizzando tecniche di Dependency Injection le quali andranno a ridurre le dimensioni del boilerplate del codice. Spring ha meccanismi per la gestione automatica di richieste parallele, è molto efficiente e la vasta community ha sviluppato molteplici plugin che ne estendono le funzionalità. Per la gestione delle dipendenze e per il deploy su App Engine abbiamo utilizzato il gestore di dipendenze Maven, il quale tramite il POM (Project Object Models) permette facilmente di rendere portabile il progetto con il minimo effort nella configurazione.

## 2.3 Google Cloud Platform

In questa sezione verranno analizzati i servizi di *Google Cloud Platform* utilizzati.

### 2.3.1 Google App Engine

Google App Engine (a.k.a. **AE**) è un PaaS (Platform as a Service) e una piattaforma di Cloud Computing per sviluppare e hostare applicazioni web gestita nei Google data center. Le applicazioni in App Engine sono sandboxed e sono eseguite su diversi server. App Engine fornisce uno scaling automatico (che viene moderato da una scaling policy) in base all'aumentare o al diminuire delle risorse necessarie. AE automaticamente alloca le risorse necessarie anche per ulteriori richieste.

App Engine supporta applicazioni scritte nei linguaggi: Java, GO, Python, PHP, NodeJS e .NET.

Google offre un piano free tier per un certo ammontare di richieste, quantità di bandwidth o ore di utilizzo continuo.

### 2.3.2 Cloud Storage

Google Cloud Storage è un servizio web di archiviazione di file online RESTful per l'archiviazione e l'accesso ai dati sull'infrastruttura di Google Cloud Platform. Il servizio combina le prestazioni e la scalabilità del cloud di Google con funzionalità avanzate di sicurezza e condivisione.

### 2.3.3 Cloud Functions

Con Cloud Functions, non ci sono server da creare, gestire o aggiornare. Le funzioni si scalano automaticamente e permettono high availability e tolleranti agli errori. Le Cloud Function sono ottime per creare backend serverless, eseguire l'elaborazione dei dati in tempo reale e creare app intelligenti.

Le cloud function quando invocate creano un'istanza la quale è dotata solo di RAM, porzione di questa ram può essere utilizzata per memorizzare dei file temporanei nel percorso `/tmp/`, come avviene per App Engine. Tuttavia questa memoria è condivisa da tutte le richieste servite da una data istanza. Per ovviare a questo problema ad ogni richiesta è assegnato un id randomico (UUID), questo id è utilizzato per creare una "sandbox" ovvero una cartella nel percorso `/tmp/ + UUID + /` così da separare gli ambienti tra le richieste che potrebbero sovrapporsi.

### 2.3.4 Firebase Auth & Accesso alle API e File



Con l'obiettivo di mettere in sicurezza l'applicazione ed evitare gli abusi abbiamo utilizzato il servizio Firebase Auth per autenticare l'accesso alle API e al frontend.

#### **API autenticate:**

Ogni chiamata alle API richiede un Authorization Header generato al login, il client si autentica con Firebase richiedendo il proprio authorization code, il quale sarà disponibile per un periodo limitato al termine di esso sarà revocato e quindi non più valido. Inoltre tramite le API di Firebase Auth calcoliamo lo User Identification Code (UID) corrispondente all'utente, il quale è di fondamentale importanza per la gestione dell'accesso ai file.

Il backend per ogni chiamata controllerà la validità dell'authorization code, se il controllo di validità fallisce la chiamata restituirà un errore all'utente, altrimenti proseguirà.

## Capitolo 3

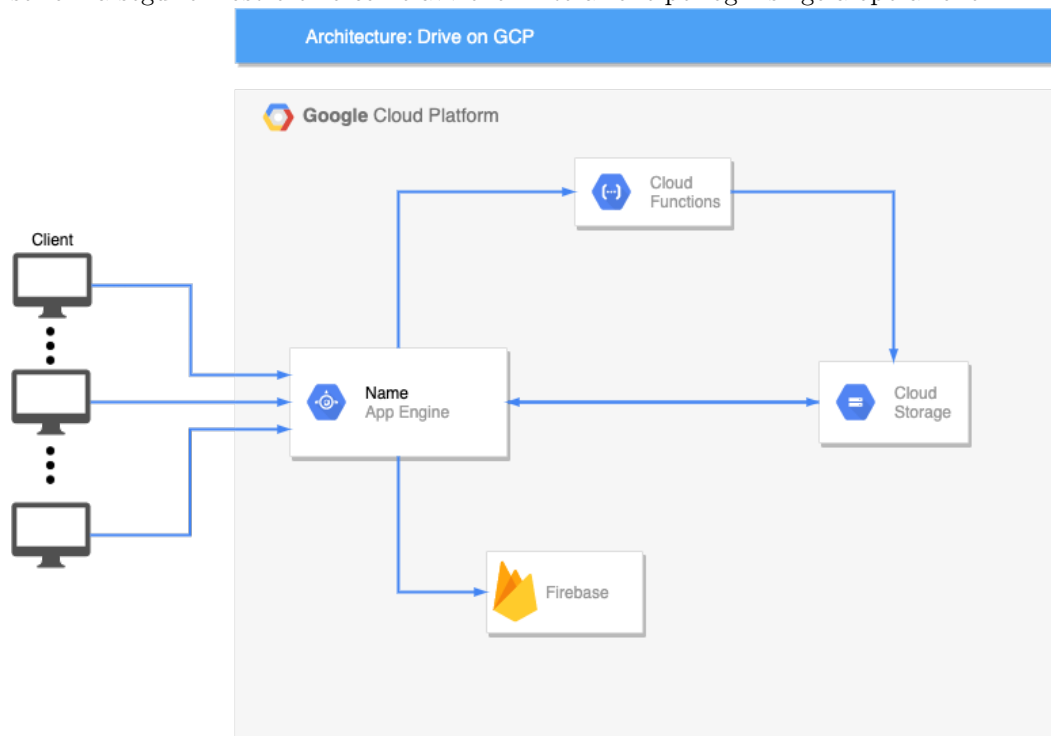
# Accesso ai file

L'accesso ai file è moderato mediante Firebase Auth, dopo aver autenticato una API calcoliamo lo UID dell'utente il quale restituisce un id di 28 caratteri, questa stringa è utilizzata come root del file system creato per l'utente. Supponiamo che l'utente voglia effettuare un upload di un file e caricarlo in una certa cartella di destinazione. L'applicazione effettua dei controlli per evitare alcuni dei più comuni problemi di sicurezza relativi all'accesso ai file system. Per isolare i drive personali di ogni utente, l'applicazione concatena il suo UID con la path della cartella di destinazione. Questa concatenazione avviene lato backend e il backend calcola autonomamente lo UID, evitando di riceverlo dal frontend, sempre per ragioni di sicurezza all'accesso.

## Capitolo 4

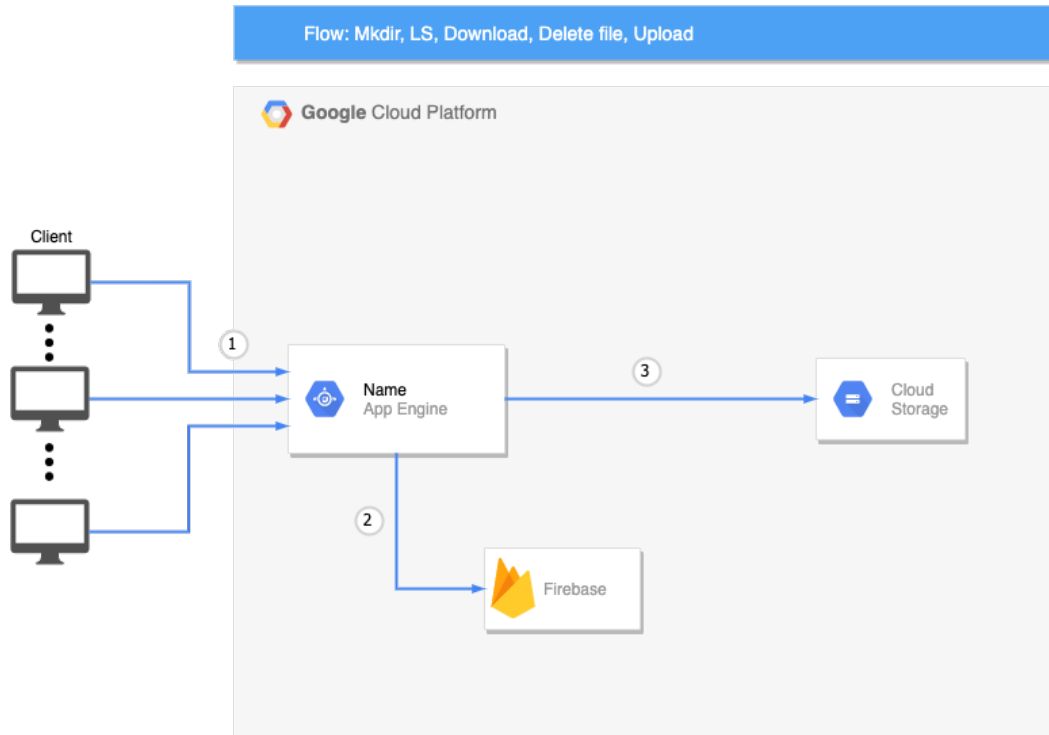
# Flusso di informazione

In questo capitolo sono illustrati i vari flussi di informazioni all'interno dell'applicazione. Evidenzieremo come le singole componenti comunicano tra di loro e in quale ordine. Nella figura in calce è possibile vedere l'architettura completa di tutto l'applicazione. Invece nelle sezioni a seguire mostreremo come avviene l'interazione per ogni singola operazione.





## 4.1 Mkdir, LS, Download, Upload



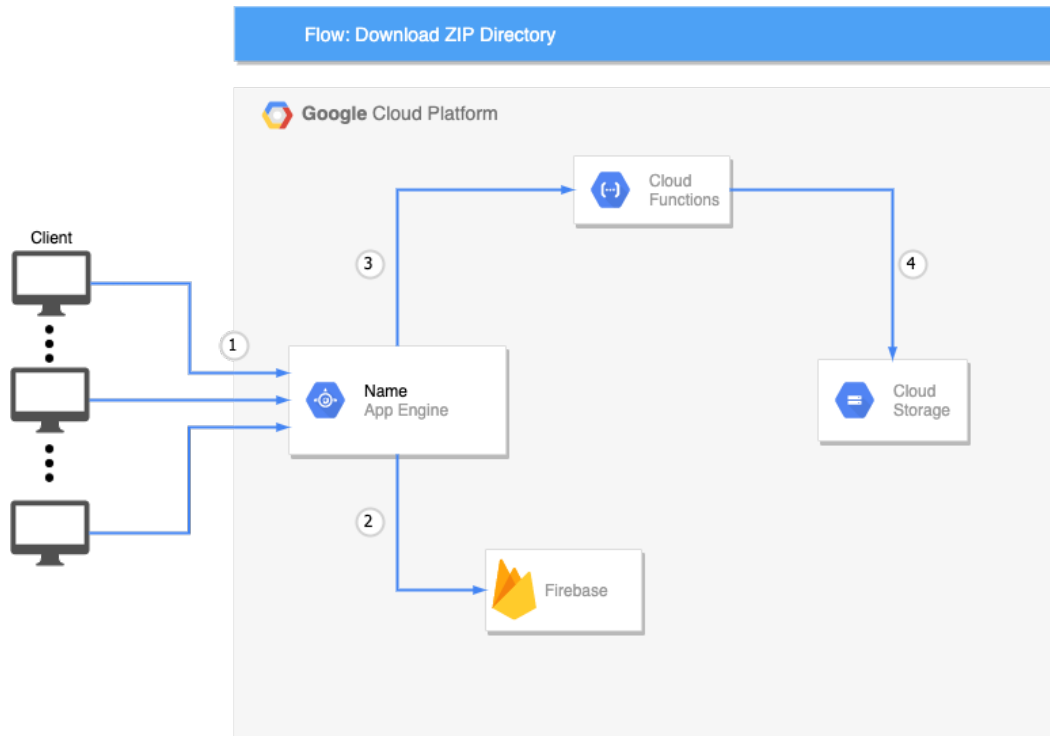
Come è possibile vedere nel grafico, le funzionalità di LS, Download di un file (non directory), la cancellazione di file/directory, la creazione di una directory e l'upload di più file assume lo stesso flusso di informazioni.

Il punto **(1)** è dove parte il flusso di informazioni, da parte di un client che esso sia tramite GUI o tramite API esposte.

Come descritto nel capitolo di Firebase ogni chiamata API è autenticata per mezzo di Firebase Auth, lato backend. Questo flusso è evidenziato nel punto **(2)** dell'immagine. Se la verifica di Firebase Auth del punto (2) fallisce, allora verrà restituito errore al client. Altrimenti si prosegue per il punto (3).

Nel punto **(3)** App Engine chiede di eseguire una delle operazioni tra cancellazione, Upload, mkdir, LS a Google Cloud Storage.

## 4.2 Download ZIP



Per scaricare una cartella abbiamo utilizzato il formato .zip essendo il più portabile tra tutti i sistemi operativi. Tuttavia in una futura estensione del progetto sarà possibile richiedere ulteriori formati di compressione come tar.gz ecc.

**Motivazioni dell'uso di una Cloud Function** Dato che App Engine non fa uso di storage persistente ma ha uno storage temporaneo nel percorso `/tmp/` il quale però andrà a consumare porzione di RAM e questo porterebbe: ad un rallentamento del sistema, un incremento dei costi per gestire file di grandi dimensioni e soprattutto porterebbe il sistema a dover generare più istanze del necessario. Inoltre questa è a seguito dei test possiamo affermare che è il download di una directory è l'operazione più dispendiosa a livello di risorse in tutta l'applicazione, per questo motivo abbiamo esternalizzato il problema affidandoci ad una Cloud Function. La Cloud function è in grado di ricreare la porzione del file system richiesta e di crearne uno zip da inviare al client.

**Per evitare gli abusi della cloud function** Il client farà richiesta al backend (App Engine + Spring) e non passerà direttamente per la Cloud Function, con il fine di evitare invocazioni inutili da parte di utenti malevoli. Quindi verranno eseguiti i passi (1), (2) e poi App Engine incapsulerà la richiesta per la function con una nuova richiesta. Questo perché altrimenti una istanza Cloud Function verrebbe generata (e pagata) anche se il Firebase token non fosse valido, invece il sistema assume che se l'utente richiede uno zip di una directory App Engine è necessariamente già attivo.

**Flow (3), (4)** La Cloud Function ha accesso diretto allo storage, esegue un download, ricostruisce la struttura della directory richiesta da zippare successivamente esegue un upload

su Cloud Storage e ne richiede un link temporaneo che poi sarà inviato al client al termine di tutta la procedura.

### **4.3 Download di un file**

Per scaricare un singolo file o una directory (.zip) il client riceverà dal backend un URL di accesso temporaneo al file presente nello storage di Google Cloud Storage. Questo permette di non far fluire i file attraverso App Engine ma di farli scaricare direttamente dall'utente permettendo una gestione più efficiente delle risorse. L'accesso a questa risorsa tramite URL è temporaneo con il fine di moderarne l'utilizzo.

## Capitolo 5

# Requirements

### 5.1 R2

#### 5.1.1 Scalability with respect to computation & high availability of the computation

Tutte le componenti utilizzate nell'applicazione sono in grado di scalare automaticamente in base al traffico e alle policy configurate. L'applicazione è stata deployed su Google App Engine, AE automaticamente gestisce le istanze, aggiungendone o rimuovendone, in base al traffico ed alla scaling policy descritta nella sezione 6.1. Inoltre il Cloud Storage, servizio di storage di GCP, è in grado di scalare automaticamente e fornire un servizio sicuro e altamente disponibile. La cloud functions è un modo di creare backend serverless e quindi per definizione è in grado di scalare autonomamente e senza limiti. La cloud function quindi come App Engine è in grado di aggiungere o rimuovere istanze in base all'utilizzo.

#### 5.1.2 Scalability with respect to data & high availability of the data

Come descritto nella sezione precedente, abbiamo utilizzato Google Cloud Storage il quale è gestito da Google Cloud Platform, ed è in grado di scalare, gestire errori e disponibilità autonomamente. Cloud Storage è un file system distribuito e ridondato e garantisce la persistenza dei dati e fault tolerance, mette inoltre a disposizione delle API con il quale è possibile far interfacciare il backend.

#### 5.1.3 Security of data in transit

Tutte le connessioni all'applicazione sono gestite solo mediante https e ogni singola operazione è gestita da API autenticate, a tutte le chiamate http viene effettuata una redirect verso https. In questo modo la connessione è crittografata e richiede un'ulteriore verifica tramite Firebase Auth per poter accedervi. Solo gli utenti registrati all'applicazione potranno superare le verifiche imposte da Firebase.

## Capitolo 6

# Esperimenti

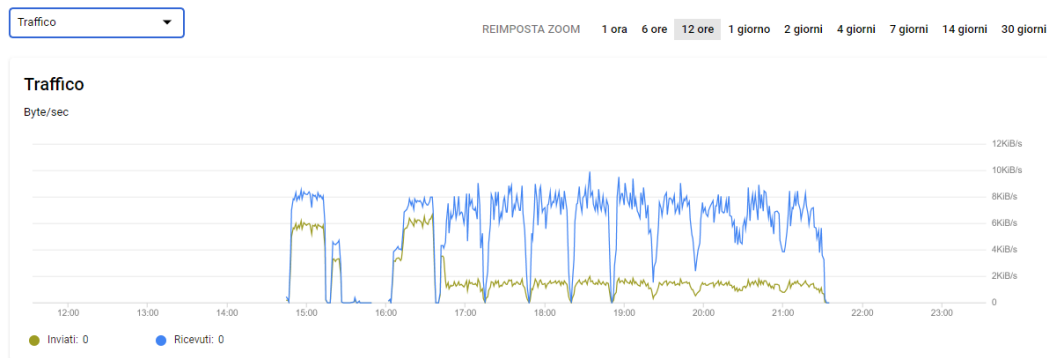
In questo capitolo vengono riportati i test che sono stati effettuati. Al fine di testare la robustezza del sistema sono stati creati e lanciati contemporaneamente diversi python script, i quali effettuano numerose chiamate ai servizi di *GCP*.

A seguito dei test, i dati per i grafici sono stati esportati utilizzando *Google Cloud Monitoring* e Grafana per la visualizzazione di alcuni grafici, sempre utilizzando *Google Cloud Monitoring* come fonte dei dati.

Le analisi riguardano principalmente *Google App Engine*, *Google Cloud Storage* e le *Cloud Functions*.

### 6.1 Test Setup

Ogni Python script genera un totale di 15.000 richieste, con picchi di 1.500 richieste ad intervalli di 5 minuti fra la fine di un picco e l'inizio del prossimo. Sono stati lanciati in parallelo 6 Python script, per un totale di 90.000 richieste con picchi di 9.000 richieste, come può essere visto nel seguente grafico.



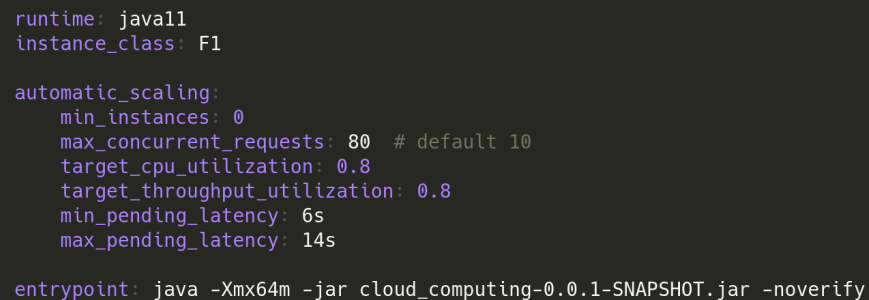
I primi due picchi sono composti da richieste ben formate, mentre gli altri picchi sono contengono richieste che porterebbero l'applicazione in errore, mettendo session token senza permessi o scaduti e richiedendo operazioni su path inesistenti. Si è voluto testare il comportamento in caso di eccezioni ed errori dato che in quel caso il flow cambia ed in molti casi la gestione di eccezioni ed errori potrebbe essere inefficiente. Con questi test è stato

possibile valutare le performance dell'architettura sia in una situazione di stress in cui tutte le risposte sono soddisfacenti ed in situazioni in cui errori ed eccezioni vengono generati da vari componenti dell'architettura.

A seguito degli esperimenti si è notato un aumento delle risorse necessarie nei picchi in cui vengono generate numerosi errori ed eccezioni.

## AE - Scaling policy & direttive di deployment

In figura possiamo vedere le scaling policy (formato .yaml) utilizzate per App Engine, abbiamo incrementato il numero massimo di richieste concorrenti, abbiamo fatto delle scelte di min e max pending latency per gestire il fatto che lo startup di time di Spring richiede diversi secondi. Inoltre abbiamo scelto F1 come `instance_class` (scegliendo tra quelle disponibili in figura 6.1) la quale è l'istanza meno costosa monetariamente e fornisce risorse più che necessarie per garantire un servizio ottimale anche durante i nostri stress test.



```
runtime: java11
instance_class: F1

automatic_scaling:
  min_instances: 0
  max_concurrent_requests: 80 # default 10
  target_cpu_utilization: 0.8
  target_throughput_utilization: 0.8
  min_pending_latency: 6s
  max_pending_latency: 14s

entrypoint: java -Xmx64m -jar cloud_computing-0.0.1-SNAPSHOT.jar -noverify
```

## 6.2 Test

Tutte le nostre scelte non hanno come obiettivo quello di fornire il servizio con le massime prestazioni, ma cercando di deployare l'applicazione con il rispetto delle performance e costi. Dato che la nostra applicazione esternalizza la maggior parte dei controlli e funzionalità facendo affidamento alle API e servizi offerti da GCP. Oltre ai test effettuati tramite script automatici abbiamo effettuato test da parte di 10 utenti reali da locazioni differenti i quali hanno utilizzato il progetto come drive personale, confermando le ottime performance ottenute dalla classe di istanza di default di App Engine.

### 6.2.1 App Engine

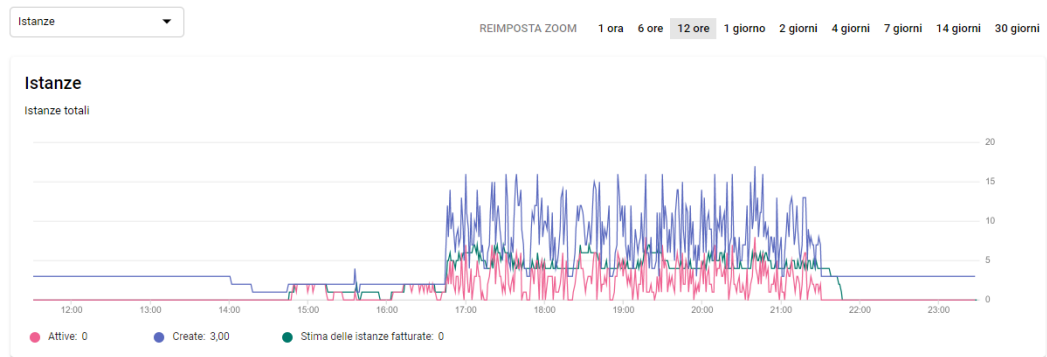
In questa sezione vengono mostrate le statistiche di App Engine, vengono mostrati i grafici di utilizzo di istanze, traffico, latenza e utilizzo memoria. È interessante notare come siano

Figura 6.1: App Engine instance classes on GCP

<div>Second gen runtimes</div> <div>First gen runtimes</div>			
Second generation runtimes are: <a href="#">Python 3</a> , <a href="#">Java 11</a> , <a href="#">Node.js</a> , <a href="#">PHP 7</a> , <a href="#">Ruby</a> , and <a href="#">Go 1.12+</a> .			
Instance Class	Memory Limit	CPU Limit	Supported Scaling Types
F1 (default)	256 MB	600 MHz	automatic
F2	512 MB	1.2 GHz	automatic
F4	1024 MB	2.4 GHz	automatic
F4_1G	2048 MB	2.4 GHz	automatic
B1	256 MB	600 MHz	manual, basic
B2 (default)	512 MB	1.2 GHz	manual, basic
B4	1024 MB	2.4 GHz	manual, basic
B4_1G	2048 MB	2.4 GHz	manual, basic
B8	2048 MB	4.8 GHz	manual, basic

state sufficienti 2 istanze attive per gestire i primi due picchi con un uso di circa 0.5 GB di memoria, mentre sono necessarie molte più risorse per gestire i picchi con errori ed eccezioni.

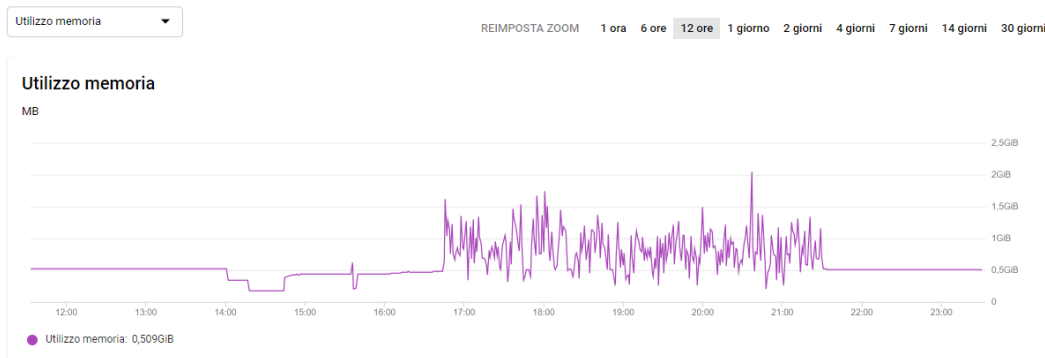
Active Instances



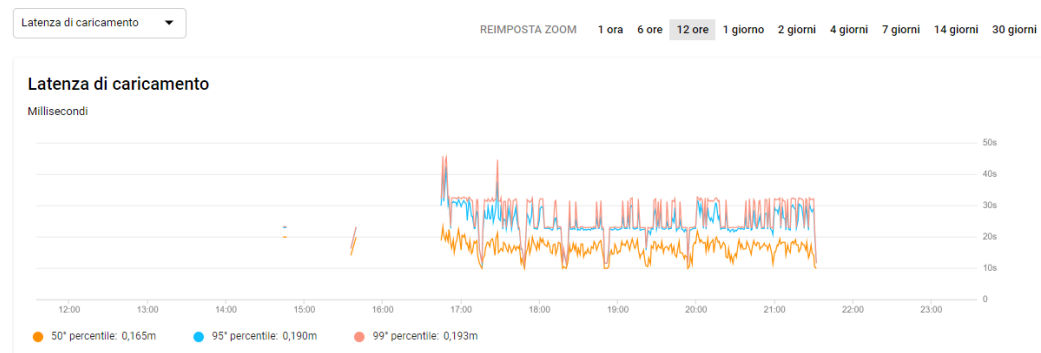
## Traffico



## Memoria



## Latenza



### 6.2.2 Google Cloud Storage

In questa sezione vengono mostrate le statistiche di Google Cloud Storage, come ci si aspettava il traffico diminuisce sostanzialmente quando sono presenti errori od eccezioni dato che



molte richieste andando in errore su App Engine non verranno mai sottomesse a Google Cloud Storage. Il picco che è possibile vedere nel grafico *Sent Byte* è rappresentato dalle chiamate alla Cloud Function per il download delle cartelle, in particolare è presente una richiesta per il download di quasi tutto il contenuto del drive.

## Request Count



## Sent Byte



## 6.2.3 Cloud Function

In questa sezione vengono mostrate le statistiche di Cloud Function, i grafici mostrano una porzione ridotta di tempo rispetto al test complessivo, questo avviene perché le richieste successive al download contenevano errori e di conseguenza è stato possibile evitare di invocare la Cloud Function risparmiando risorse.

## Active Instances



## Response Time



Executions



## Capitolo 7

# Conclusioni

Gli esperimenti hanno soddisfatto le nostre attese, App Engine ha scalato le istanze come previsto mentre Google Cloud Storage ha gestito il traffico senza problemi. L'applicazione sviluppata può essere tranquillamente espansa con ulteriori funzionalità senza il rischio di andare offline, dato che Google Cloud Platform si occupa di garantire disponibilità e performance anche in situazioni molto stressanti per l'applicazione. Come sviluppi dell'architettura potrebbe essere interessante provare sistemi di caching come Memorystore.

### 7.1 Sezione differenze con la proposta

In questa sezione vengono elencate le differenze rispetto alla proposta di progetto.

**Cloud Functions.** Nella proposta di progetto non era presente la Cloud Functions, come motivato nei capitoli precedenti abbiamo trovato utile il suo utilizzo.

**Unit Testing.** Con il fine di dividere l'applicazione (backend) in piccoli componenti e testarle singolarmente abbiamo utilizzato la tecnica di unit testing. Siamo partiti con dei test con JUnit per poi spostarci testando i controller (rest API) tramite unit testing effettuati tramite un Python script.

**Firebase Frontend.** Nella proposta si era previsto di deployare il frontend su *Firebase* mentre nell'applicativo finale il frontend è stato inserito come file statici di Spring e deployato insieme ad esso su *App Engine*. Questo è stato fatto per semplificare il deploy e la consegna del progetto, dato che non impatta né sulle performance né sui requisiti R2.