

# GCPilot

## IaC-LLM – Natural Language to Infrastructure-as-Code Generator for GCP

Danilo Cavallini, Lorenzo Di Ubaldo, Giacomo Grilli, Mattia Pasquali

<b>INTRODUZIONE</b>	<b>2</b>
<b>OBIETTIVI PROGETTO</b>	<b>2</b>
<b>SERVIZI GOOGLE CLOUD PLATFORM</b>	<b>3</b>
<b>TECNOLOGIE UTILIZZATE</b>	<b>4</b>
Frontend	4
Backend	4
<b>PIPELINE CI/CD E DEVSECOPS</b>	<b>5</b>
1. Build Docker	5
2. Test e Scansione	5
3. Deploy e Registry	5
Analisi di Sicurezza con Trivy	5
Logging e Audit	6
<b>LOGIN E VALIDAZIONE</b>	<b>6</b>
<b>ARCHITETTURA APPLICAZIONE</b>	<b>7</b>
<b>WORKFLOW DELL'APPLICAZIONE</b>	<b>8</b>
<b>STORAGE, BUILD E RUN</b>	<b>13</b>
Cloud Storage	13
Cloud Build	13
Cloud Run	14
<b>GCLOUD NAT</b>	<b>14</b>
<b>LOAD BALANCER</b>	<b>15</b>
<b>SYSTEM RESILIENCE</b>	<b>15</b>
<b>LLM MODEL</b>	<b>16</b>
<b>MODEL EVALUATION</b>	<b>17</b>
<b>TEST SCALABILITA' E PERFORMANCE</b>	<b>19</b>
Scalability Test	19
Spike Test	22
Stress Test	23
<b>COST ESTIMATION</b>	<b>23</b>
<b>COSTI DI MANTENIMENTO</b>	<b>24</b>
<b>CONCLUSIONI</b>	<b>25</b>
<b>BIBLIOGRAFIA</b>	<b>25</b>

## **INTRODUZIONE**

In un contesto tecnologico caratterizzato da una crescente adozione di soluzioni cloud, l'automazione dell'infrastruttura (Infrastructure as Code – IaC) rappresenta un elemento essenziale per garantire scalabilità, ripetibilità e sicurezza nella gestione delle risorse IT. La crescente complessità delle architetture cloud rende tuttavia necessario ridurre il livello di competenze tecniche richieste per la progettazione e il provisioning delle infrastrutture.

In questo scenario, l'impiego di Large Language Models (LLM) si configura come un'opportunità per semplificare la creazione di ambienti infrastrutturali complessi. Tali modelli, opportunamente integrati in un sistema di generazione automatica, permettono di trasformare prompt in linguaggio naturale in codice di configurazione Terraform valido e sicuro, pronto per essere distribuito su Google Cloud Platform (GCP).

## **OBIETTIVI PROGETTO**

Il progetto ha avuto come obiettivo principale la realizzazione di un servizio in grado di generare configurazioni infrastrutturali automatizzate per Google Cloud Platform a partire da descrizioni in linguaggio naturale fornite dall'utente. Il sistema supporta componenti cloud diffusi come VM, servizi serverless, database, reti e storage, generando efficacemente configurazioni Terraform pronte al deployment.

L'interfaccia web progettata consente agli utenti di:

- Inserire il prompt descrittivo dell'infrastruttura desiderata;
- Visualizzare e modificare il codice generato;
- Validarne la correttezza tramite strumenti nativi di Terraform (validate, plan);
- Effettuare il deployment dell'infrastruttura su GCP (apply) o la sua rimozione (destroy);
- Salvare le sessioni per consultazioni o riutilizzi futuri.

## **SERVIZI GOOGLE CLOUD PLATFORM**

L'architettura del sistema si basa sull'integrazione di diversi servizi Google Cloud Platform (GCP), ognuno con un ruolo specifico all'interno del flusso operativo:

- Vertex AI: impiegato per l'inferenza del Large Language Model (LLM), responsabile della generazione automatica del codice Terraform;
- Cloud Run: ospita il backend applicativo e funge da orchestratore per le richieste utente, integrando i vari componenti del sistema;
- Cloud Build: utilizzato per eseguire pipeline CI/CD, applicare le configurazioni Terraform in modo automatizzato e garantire la coerenza dei deployment;
- Cloud Storage: utilizzato per la persistenza dei file di configurazione, snapshot di codice, log delle sessioni e risultati dei deployment;
- IAM (Identity and Access Management): gestisce i permessi e i ruoli, garantendo un accesso sicuro e controllato alle risorse cloud, con isolamento tra gli utenti;
- Cloud Logging: centralizza la raccolta dei log generati durante le varie fasi del processo (generazione, validazione, deploy), consentendo il monitoraggio e il debugging delle operazioni.

## TECNOLOGIE UTILIZZATE

Per andare a sfruttare l'ecosistema di Google Cloud Platform, sempre gestito tramite l'uso di Terraform lato infrastrutturale, sono stati realizzati frontend e backend tramite l'utilizzo delle principali tecnologie:

### Frontend

- React: la libreria JavaScript è stata utilizzata per sviluppare un'interfaccia utente dinamica e reattiva. L'approccio component-based ha facilitato la modularizzazione dell'interfaccia e l'implementazione di funzionalità come l'editor del codice e la visualizzazione del piano Terraform.
- TypeScript: l'utilizzo di TypeScript ha introdotto tipizzazione statica e maggiore robustezza del codice lato client, migliorando la manutenibilità del progetto.
- Vite: impiegato come build tool per lo sviluppo del frontend, ha permesso tempi di compilazione rapidi e una configurazione semplificata rispetto a soluzioni tradizionali.

### Backend

- Node.js: l'ambiente di esecuzione JavaScript ha fornito una base leggera e performante per la logica server-side del sistema, facilitando l'integrazione con API esterne e servizi GCP. Node.js risulta ottimo per il nostro caso in quanto con la sua architettura non bloccante risulta ottima nella gestione di input e output, inoltre non necessitiamo di eseguire lunghe operazioni CPU bound che potrebbero risultare poco scalabili anche a causa della natura del linguaggio Javascript e dello scarso livello di parallelizzazione offerto da Node. La maggior parte del carico computazionale risiede infatti nell'inferenza del Large Language Model che viene delegata alla Vertex Api messa a disposizione da Google.
- TypeScript: anche lato backend, l'utilizzo di TypeScript ha garantito una maggiore sicurezza del codice, attraverso il controllo dei tipi, e ha reso più chiara la definizione delle interfacce tra i moduli.
- Docker: utilizzato per il packaging e la containerizzazione dell'applicazione. Consente di definire in modo dichiarativo l'ambiente di esecuzione, rendendo il servizio portabile, facilmente distribuibile e compatibile con i requisiti di esecuzione di Cloud Run. La containerizzazione, permette l'isolamento, la riproducibilità e la scalabilità del backend.

## PIPELINE CI/CD E DEVSECOPS

Nel progetto abbiamo adottato pratiche DevSecOps, fondamentali per garantire efficienza, affidabilità e sicurezza del ciclo di vita software.

- DevOps mira ad automatizzare tutte le fasi dello sviluppo dalla build al test fino al deployment per favorire una consegna continua, tracciabile e stabile.
- DevSecOps estende questo approccio integrando la sicurezza fin dalle prime fasi, includendo analisi delle vulnerabilità e audit continuo come parte integrante del processo.

Tutta la logica di build, test e deploy è orchestrata da Cloud Build, con configurazione in "cloudbuild.yaml":

### 1. Build Docker

- Utilizzo di `gcr.io/cloud-builders/docker` per creare immagini basate su Dockerfile.
- Tag dinamico `${SHORT_SHA}` per versioning e audit.

### 2. Test e Scansione

- `npm test` su frontend e backend (fallback non bloccante).
- Scansione vulnerabilità con Trivy (`ghcr.io/aquasecurity/trivy`) su livelli HIGH/CRITICAL.

### 3. Deploy e Registry

- Push delle immagini su Artifact Registry.
- Deploy automatizzato su Cloud Run dei due servizi (via `gcloud run deploy`), con variabili d'ambiente (`GOOGLE_PROJECT_ID`, `JWT_SECRET`, ecc.) e connettore VPC per egress/ingresso.

E contiene i seguenti step per analisi vulnerabilità e logging:

#### Analisi di Sicurezza con Trivy

- Utilizzo di Trivy (`ghcr.io/aquasecurity/trivy`) per l'analisi delle vulnerabilità nell'immagine Docker

- Analisi su livelli di gravità HIGH e CRITICAL
- Non blocca la pipeline ma segnala gli eventuali rischi nel log (scan non-bloccante)
- Conformità alle best practice DevSecOps per "shift-left security"

## Logging e Audit

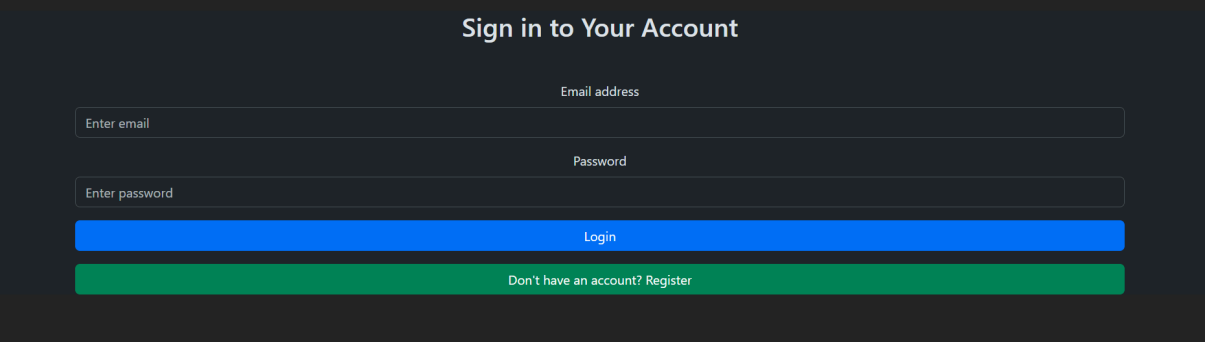
- Tutti i log sono conservati in Cloud Logging tramite 'CLOUD\_LOGGING\_ONLY'
- Permette audit trail, debug e controllo centralizzato delle build

Con questa configurazione la pipeline CI/CD garantisce rilascio continuo e tracciabile, controllo di qualità e sicurezza proattiva, e integrazione fluida nel ciclo di sviluppo.

## LOGIN E VALIDAZIONE

Il sistema di autenticazione è progettato per operare in un contesto cloud-native, sfruttando i meccanismi di sicurezza offerti da Google Cloud Platform (GCP). L'architettura si fonda su tre elementi principali: la gestione delle credenziali utente, l'autenticazione del backend verso Firestore e un middleware di validazione per le API protette.

Per garantire una comunicazione sicura con Firestore, l'applicazione backend utilizza un Service Account configurato in GCP IAM. Questo Service Account è associato a una chiave JSON, caricata dal backend all'avvio. La chiave funge da file di credenziali, consentendo al backend di autenticarsi automaticamente verso le API di Firestore senza richiedere interazioni manuali. Tale approccio garantisce che solo l'applicazione server-side possa accedere alle operazioni di lettura e scrittura sui dati degli utenti, riducendo il rischio di esposizione accidentale.



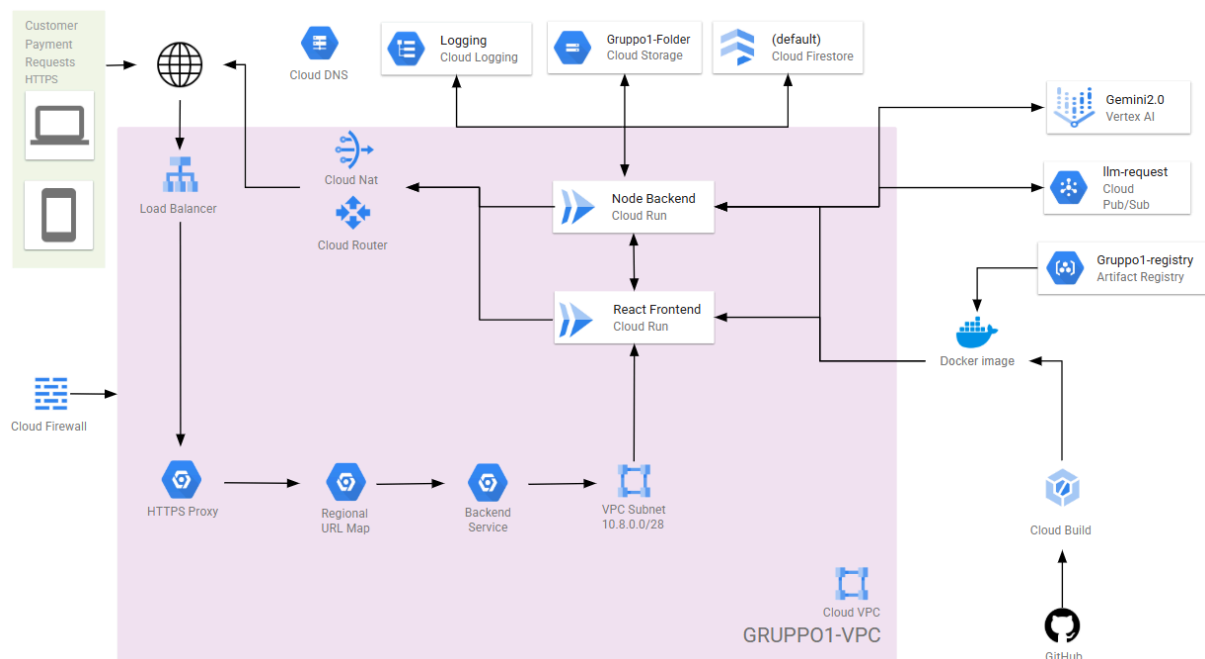
The image shows a login form with a dark background. At the top, the title "Sign in to Your Account" is centered in white. Below the title, there are two input fields: the first is labeled "Email address" and contains the placeholder text "Enter email"; the second is labeled "Password" and contains the placeholder text "Enter password". Below these fields are two buttons: a blue "Login" button and a green "Don't have an account? Register" button.

Durante la registrazione, le password degli utenti vengono sottoposte a hashing mediante l'algoritmo bcrypt (con salt), e il risultato viene memorizzato in Firestore insieme alla mail. Firestore funge da archivio centralizzato per le credenziali, beneficiando delle sue caratteristiche di alta disponibilità e scalabilità automatica.

L'accesso alle API protette è gestito da un middleware, che estrae e valida i token JWT dai cookie inviati dal client. Questo meccanismo assicura che solo gli utenti autenticati possano accedere alle risorse protette. La scelta di un'architettura stateless, basata su JWT e cookie sicuri, la rende particolarmente adatta a deployment in ambienti containerizzati e scalabili.

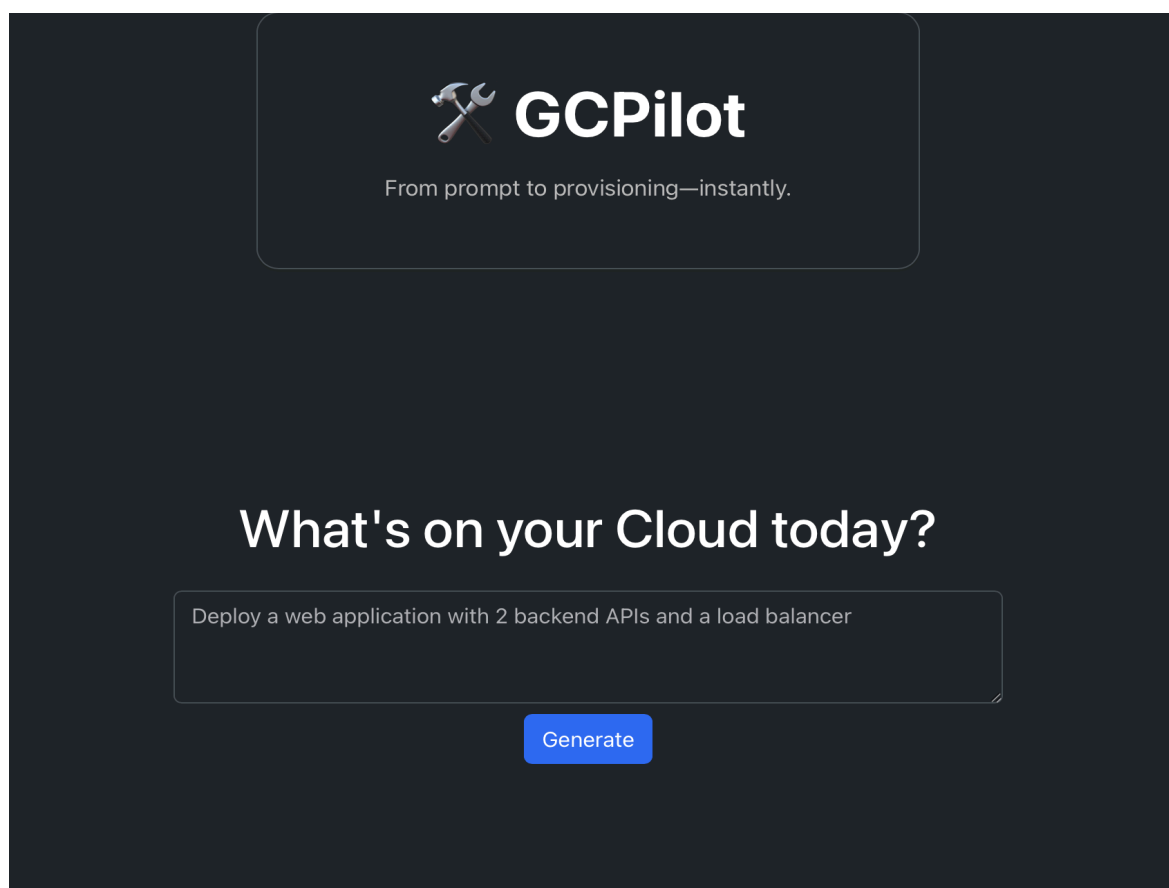
## ARCHITETTURA APPLICAZIONE

- Frontend React (Cloud Run + HTTPS Load Balancer) gestisce l'input utente e ospita l'editor Terraform.
- Backend Node.js deployato su Cloud Run (stateless) espone API REST per “generate”, “preview”, “apply” e “destroy”.
- Per la generazione del codice il backend pubblica i prompt su Pub/Sub (IIm-requests), e un subscriber innesca la chiamata a Vertex AI, salvando prompt e risposte su Cloud Logging.
- Le configurazioni .tf modificate vengono archiviate su Cloud Storage in percorsi isolati per utente (gs://<bucket>/<email>/<folderId>/...).
- Le operazioni di terraform plan, apply e destroy sono eseguite in container isolati via Cloud Build, che restituisce al frontend gli output e li traccia anch'essi su Cloud Logging.
- IAM protegge e gestisce gli accessi e le API, garantendo isolamento tra sessioni utente.



## WORKFLOW DELL'APPLICAZIONE

Una volta effettuato l'accesso, l'utente viene accolto dalla dashboard di GCPilot, l'interfaccia web progettata per guidare l'intero processo di generazione e gestione dell'infrastruttura. Nella schermata principale è disponibile una textbox dove l'utente può inserire una descrizione in linguaggio naturale dell'infrastruttura desiderata. Una volta definito il prompt, l'utente avvia la generazione premendo il pulsante "Generate".



Il frontend esegue una chiamata HTTPS POST al backend Node.js, trasmettendo il contenuto del prompt insieme alle informazioni di sessione dell'utente.

Il backend, eseguito in Cloud Run all'interno di un container Docker, riceve la richiesta e la gestisce attraverso una serie di passaggi asincroni orchestrati con Pub/Sub:

- Il backend pubblica il messaggio (contenente il prompt e i metadati) su un topic Pub/Sub.
- Un servizio in ascolto tramite una subscription elabora i messaggi pubblicati, consentendo una gestione scalabile e asincrona delle richieste.

Non appena il backend riceve il messaggio dalla subscription, il contenuto del prompt viene registrato su Cloud Logging per finalità di auditing e debugging. Successivamente, il backend



invia il prompt a Vertex AI, sfruttando un Large Language Model (LLM) per la generazione del codice Terraform.

Vertex AI elabora la descrizione in linguaggio naturale e restituisce un file di configurazione Terraform in formato HCL (HashiCorp Configuration Language). Anche la risposta generata viene salvata su Cloud Logging, garantendo così la tracciabilità completa dell'intero processo.

Il codice Terraform generato viene inviato dal backend al frontend e visualizzato in un editor integrato nell'interfaccia utente. L'utente ha la possibilità di:

- Analizzare il codice.
- Modificarlo secondo necessità per adattarlo ai suoi bisogni.
- Confermarlo per proseguire con le fasi successive.

## Terraform Code

```
1 terraform {
2   required_providers {
3     random = {
4       source = "hashicorp/random"
5       version = "3.1.0"
6     }
7   }
8 }
9
10 resource "random_string" "random" {
11   length      = 8
12   special     = false
13   override_special = "/@f$"
14 }
15
16 output "random-uuid" {
17   value = uuid()
18 }
19
```

 Confirm Code

Una volta confermato, il backend salva la configurazione all'interno di una directory utente dedicata su Cloud Storage, garantendo isolamento e persistenza dei dati per sessioni future.

Per garantire la correttezza e la sicurezza della configurazione, l'utente può avviare tramite un semplice pulsante una fase di preview sfruttando i comandi nativi di Terraform:

- Il frontend invia una richiesta al backend per eseguire un terraform plan.
- Il backend, tramite Cloud Build, avvia un container isolato che applica il comando terraform plan sulla configurazione salvata.
- L'output del piano viene restituito al frontend e visualizzato per consentire all'utente di verificare le risorse che saranno create, modificate o distrutte.

► Run Preview

### Terraform Plan Preview

```

==== TERRAFORM INIT ====
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/random from the dependency lock file
- Using previously-installed hashicorp/random v3.1.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.

```

Se l'utente valida la configurazione proposta premendo su “Run Apply”, viene avviata la fase di deployment:

- Il frontend invia una richiesta di terraform apply al backend.
- Anche in questo caso, il backend esegue il comando in un container tramite Cloud Build per garantire un ambiente sicuro e isolato.
- L'output del processo di applicazione viene inoltrato all'interfaccia utente e registrato su Cloud Logging.

### Terraform Apply

```

Successfully configured the backend "gcs"! Terraform will automatically
use this backend unless the backend configuration changes.

Initializing provider plugins...
- Finding hashicorp/random versions matching "3.1.0"...
- Installing hashicorp/random v3.1.0...
- Installed hashicorp/random v3.1.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.

==== TERRAFORM PLAN ====

```

► Run Apply

L'utente può tramite il “Run Destroy”, eseguire anche un terraform destroy, tramite apposito comando per rimuovere completamente l'infrastruttura precedentemente creata.

```
+ min_special = 0
+ min_upper   = 0
+ number      = true
+ override_special = "/@£$"
+ result      = (known after apply)
+ special     = false
+ upper       = true
}

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
+ random-string = (known after apply)
+ random-uuid   = (known after apply)
random_string.random: Creating...
random_string.random: Creation complete after 0s [id=7GrRzeL6]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

random-string = "7GrRzeL6"
random-uuid   = "ee902bad-476f-095a-55dc-83a43aab96e9"
```

► Run Destroy

L'utente può inoltre in ogni momento recuperare le precedenti conversazioni tramite apposito menù a tendina avendo così sempre accesso al prodotto generato.

### Previous Chats

User Request: Deploy a web application with a frontend and backend using Docker and Kubernetes.

Load .tf Output

User Request: Set up a Pub/Sub topic named email-events

Load .tf Output

User Request: I need a public Cloud Storage bucket to upload images


Load .tf Output

User Request: Provision a Cloud SQL instance with MySQL

Load .tf Output

User Request: Create a Cloud Function triggered by HTTP requests

Load .tf Output

**GCPilot**

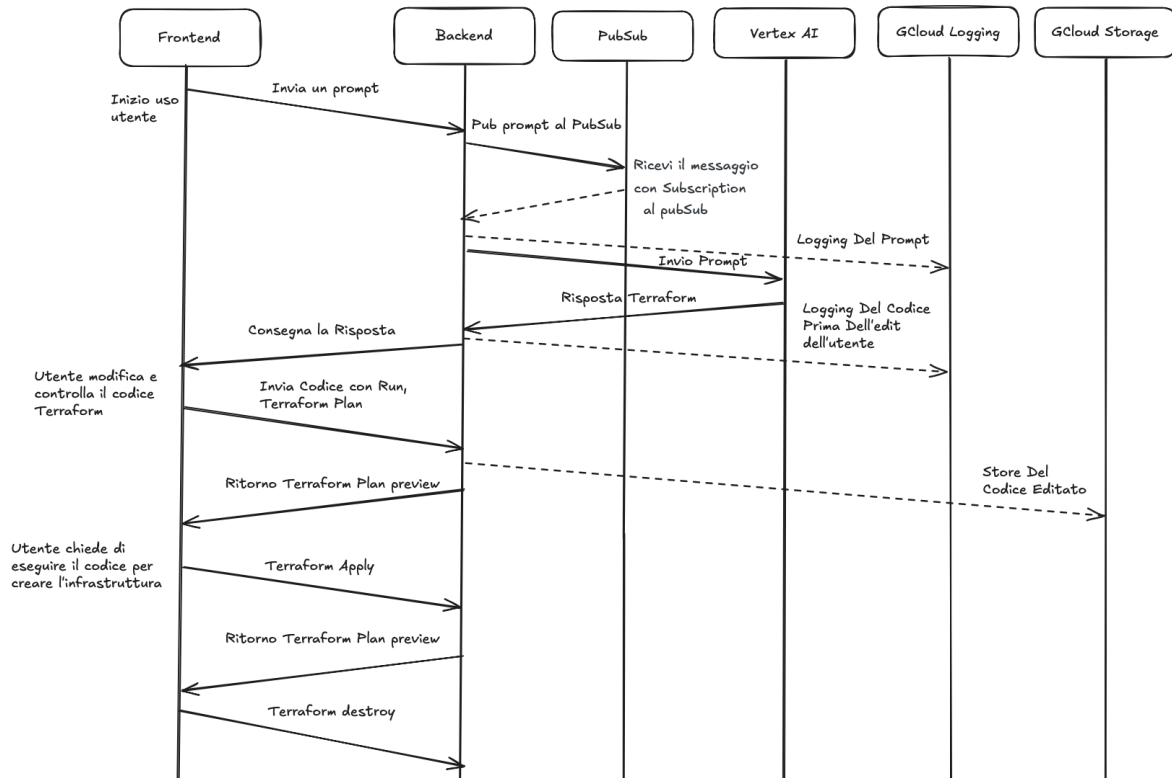
From prompt to provisioning—instantly.

## What's on your Cloud today?

Deploy a web application with 2 backend APIs and a load balancer

Generate

Segue uno schema riassuntivo del funzionamento dell'applicazione suddivisa per i diversi ambienti e/o servizi:



## STORAGE, BUILD E RUN

### Cloud Storage

Il sistema utilizza un bucket Google Cloud Storage centralizzato per la gestione persistente dei file Terraform generati. Ogni utente dispone di un percorso dedicato e isolato, nel formato `gs://<bucket>/<email>/<folderId>/`, dove vengono salvati:

- i file .tf prodotti da Vertex AI (inclusi moduli, main.tf e variabili personalizzate);
- le versioni modificate manualmente dall'utente;
- i file di stato e gli snapshot degli output Terraform.

Le operazioni di scrittura e lettura su GCS sono gestite dal backend tramite la libreria `@google-cloud/storage`, attraverso le funzioni `uploadTerraform` e `listTerraformFiles`, che garantiscono integrità e riusabilità dei file nel tempo.

### Cloud Build

La piattaforma sfrutta Cloud Build per eseguire in ambienti containerizzati e isolati tutte le operazioni Terraform richieste dall'utente, in particolare `terraform plan`, `apply` e `destroy`.

Le build sono definite nel file `cloudbuild.yaml` e si articolano in più fasi, tra cui:

- la build dell'immagine Docker tramite il builder ufficiale (`gcr.io/cloud-builders/docker`);
- il deploy del backend su Cloud Run (`gcr.io/cloud-builders/gcloud`);
- l'esecuzione dei comandi Terraform su configurazioni montate direttamente dal bucket GCS, sfruttando variabili come `GOOGLE_SHARED_USER_BUCKET` e `GOOGLE_PUBSUB_TOPIC` per adattare dinamicamente la build alla sessione utente.

In questo modo ogni invocazione di Terraform avviene in un ambiente effimero, sicuro e tracciabile, con output registrati in Cloud Logging.

## Cloud Run

L'infrastruttura applicativa è ospitata interamente su Cloud Run, con due servizi distinti:

- react-frontend: ospita l'interfaccia utente;
- node-backend: espone API REST e gestisce la logica applicativa lato server.

Entrambi i servizi sono definiti in Terraform tramite risorse `google_cloud_run_v2_service` e sono configurati per scalare automaticamente in base al carico. Sono impostati rispettivamente con un massimo di 10 istanze per il frontend e 6 per il backend.

L'accesso al backend è esposto tramite l'opzione `INGRESS_TRAFFIC_INTERNAL_ONLY`, che consente solamente alla VPC interna di accedere al backend, in questo modo il frontend può raggiungere il backend in modo completamente sicuro. Inoltre, è stato configurato un connettore VPC (`google_vpc_access_connector`) per permettere l'accesso in uscita a risorse interne della VPC, come Artifact Registry o altri servizi GCP privati.

Il Frontend è esposto tramite l'opzione `INGRESS_TRAFFIC_INTERNAL_LOAD_BALANCER` in questo modo il frontend può essere raggiunto dall'esterno solo tramite l'indirizzo specificato nel HTTPS Load Balancer, in oltre il servizio ha l'opzione "allow unauthenticated" aggiunge a tutti gli utenti il IAM role `run.invoker` in modo che possano usare liberamente il frontend.

Il Backend utilizza i servizi di GCloud Pub/Sub, GCloud Storage, GCloud Firestore, GCloud Vertex AI, per chiamare questi servizi si utilizzano dei service account appositi, in modo che ogni chiamata sia limitata dagli IAM roles di quel specifico service account.

Questi servizi devono avere una connessione alle api di google per funzionare, dato che il servizio backend è all'interno di un Docker container e all'interno del Google VPC, bisogna garantire l'accesso a internet per questi servizi. Per fare ciò usiamo Gcloud Nat

## GCLOUD NAT

Cloud Nat è un servizio di Google Cloud che consente alle istanze nella tua rete Virtual Private Cloud (VPC) di accedere a Internet senza bisogno di indirizzi IP pubblici. Sinteticamente si può dire che permette di tradurre gli indirizzi IP privati delle tue istanze in indirizzi IP pubblici quando avviano connessioni online.

## LOAD BALANCER

Per gestire il traffico in ingresso verso il frontend, è stato configurato un HTTPS Load Balancer esterno tramite Google Cloud Load Balancing. Questo componente è responsabile dell'instradamento delle richieste HTTPS degli utenti verso l'interfaccia web del sistema, esposta come servizio Cloud Run.

Il bilanciatore è stato implementato in modalità serverless, utilizzando una Serverless Network Endpoint Group (NEG) come backend. Tale NEG collega il Load Balancer al servizio Cloud Run che ospita il frontend, consentendo di integrare facilmente l'infrastruttura serverless con la rete VPC.

Il traffico HTTPS viene quindi gestito tramite:

- una regola di forwarding globale che intercetta le richieste sulla porta 443;
- un proxy HTTPS target che inoltra le richieste a una URL Map;
- la URL Map che associa i percorsi al Backend Service connesso al Cloud Run frontend.

Questa architettura garantisce:

- scalabilità automatica, grazie all'integrazione con Cloud Run;
- disponibilità globale, grazie all'uso di IP globali e del bilanciamento geografico;
- maggiore sicurezza e flessibilità, potendo aggiungere regole di routing e filtraggio.

L'infrastruttura del Load Balancer è interamente definita tramite Terraform, rendendo il setup riproducibile, modulare e versionabile.

## SYSTEM RESILIENCE

Google Cloud Run gestisce la tolleranza ai guasti in modo nativo attraverso un'infrastruttura serverless altamente resiliente e distribuita. Ogni servizio Cloud Run è automaticamente replicato in più zone all'interno della regione selezionata, garantendo l'alta disponibilità anche in caso di guasti localizzati. In caso di errore di un'istanza (ad esempio per crash o malfunzionamento del container), Cloud Run può automaticamente avviare nuove istanze per sostituire quelle non operative, senza necessità di intervento manuale.

Secondo l'SLA, per le versioni non-GPU distribuite su zone multiple, Cloud Run offre un uptime mensile del 99,95 %, traducibile in non più di 21,9 minuti di inattività al mese. Questo livello di servizio garantisce che il carico di lavoro rimanga operativo anche in presenza di errori infrastrutturali o fault di singole istanze.

L'uso di Pub/Sub per la comunicazione asincrona tra i servizi contribuisce a disaccoppiare i componenti e ad aumentare la tolleranza ai guasti, consentendo il buffering dei messaggi in caso di rallentamenti o malfunzionamenti temporanei di uno dei consumer.

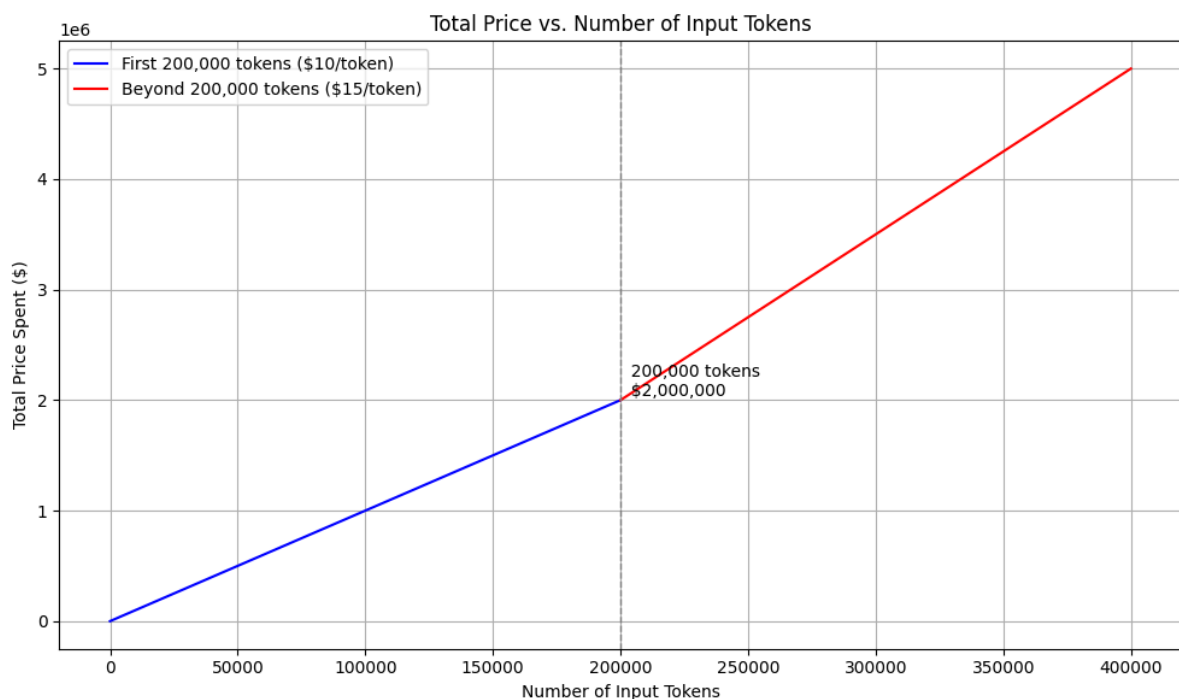
## LLM MODEL

I Large Language Models (LLM) rappresentano una delle innovazioni più significative nell'ambito dell'intelligenza artificiale applicata all'elaborazione del linguaggio naturale. Questi modelli, basati sull'architettura Transformer, sono in grado di comprendere e generare testo con elevata coerenza, sfruttando l'addestramento su grandi quantità di dati testuali provenienti da fonti eterogenee. GCPilot sfrutta l'abilità degli LLM di comprendere l'intenzione dell'utente, espressa in linguaggio naturale e restituire una risposta coerente con la richiesta dell'utente.

Abbiamo scelto di utilizzare i modelli Gemini in quanto offrono un modello di pagamento a consumo in base ai token di input e di output più conveniente rispetto ad affittare una GPU per ospitare un modello di terze parti.

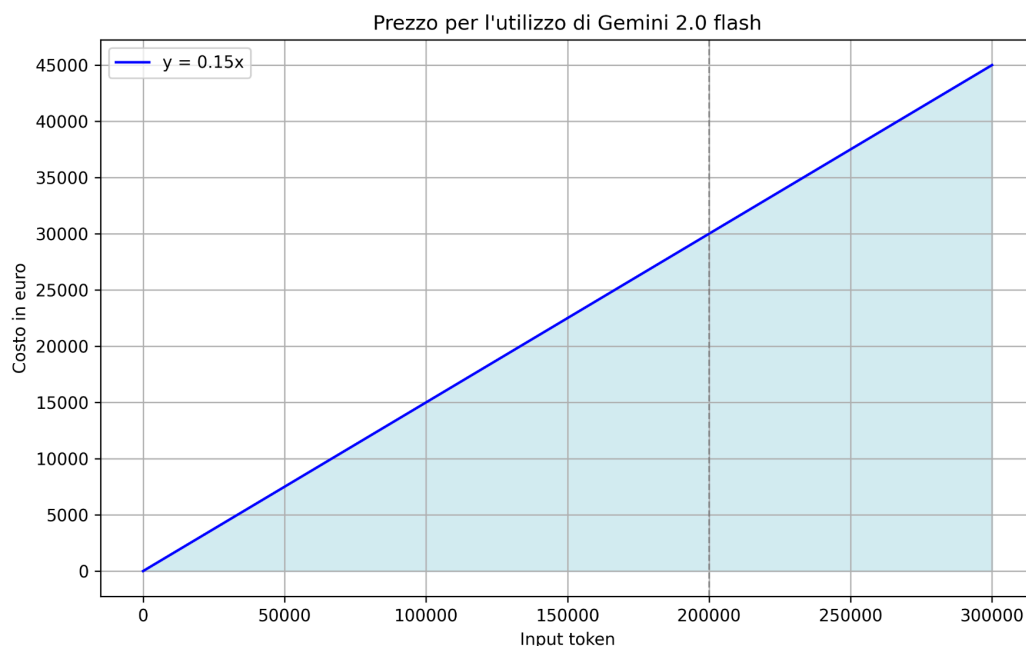
I modelli che Google mette a disposizione sono: Gemini 2.5 Pro, Gemini 2.0 Flash, Gemini 2.0 Flash-Lite lasciamo quindi fuori dalla valutazione Gemini 2.5 Flash che alla data di scrittura di questo documento sono disponibili nella forma di "private general availability offering" con frequenti cambi di endpoint e quindi inadatti per il rilascio di un software in produzione.

Nell'immagine possiamo osservare ad esempio le modalità di billing per l'utilizzo di Gemini 2.5 pro, che aumentano all'aumentare dei token di input.





Il grafico sottostante invece raffigura il prezzo per l' utilizzo di Gemini 2.0 Flash, che al compromesso di performance minori ha un migliore rapporto token per euro spesi.



Risulta evidente che è necessario uno studio delle performance dei diversi modelli sul dominio del problema da risolvere per assicurarsi di non sprecare soldi su un modello di cui non si usano appieno tutte le capacità.

## MODEL EVALUATION

Per la valutazione dei diversi modelli abbiamo utilizzato il dataset IaC-Eval che offre un elenco di prompt e corrispettivi output sotto forma di file Terraform. Possiamo così confrontare la risposta effettiva di ogni modello con la risposta desiderata e valutarla secondo alcune metriche per il confronto di testi: BLUE e ROUGE.

BLEU è una metrica originariamente sviluppata per valutare la qualità della traduzione automatica, ma viene ampiamente utilizzata per confrontare testi generati automaticamente con testi di riferimento. Si basa sul confronto di n-grammi (sequenze di parole) e misura la precisione, ovvero quante porzioni del testo generato coincidono con quelle attese. Per quanto riguarda il nostro caso d'uso, BLEU è utile per verificare quanto il codice prodotto ricalchi fedelmente quello di riferimento in termini di sintassi e struttura locale.

ROUGE è una famiglia di metriche nate per la valutazione automatica del riassunto testuale. A differenza di BLEU, ROUGE si concentra sul recall, cioè sulla quantità di contenuto corretto che il modello è riuscito a generare rispetto al riferimento. Le varianti più comuni sono ROUGE-1 (unigrammi), ROUGE-2 (bigrammi) e ROUGE-L (sottosequenze comuni più lunghe). Nell'ambito dei file Terraform, ROUGE aiuta a misurare quanto il contenuto prodotto sia completo e semanticamente coerente rispetto all'output atteso.

Si è poi utilizzata per l'inferenza delle risposte dei vari modelli, la tecnica del few-shot prompting, in questo approccio si forniscono pochi esempi di domande e risposte all'interno del prompt per guidare il comportamento del modello. Questi esempi mostrano come trasformare un input in un output desiderato, permettendo al modello di generalizzare e produrre risposte simili per nuovi input, senza bisogno di ulteriore addestramento. È particolarmente utile quando si ha poco testo di riferimento, ma si vuole comunque ottenere una generazione coerente e controllata.

Se per esempio volessimo far convertire al modello un elenco puntato a un testo in formato json il prompt potrebbe essere scritto così:

Converti in formato JSON un elenco puntato di seguito troverai esempi di input e output, alla fine troverai la domanda a cui dovrai rispondere.

Esempi:

Input: “- Nome: Luca, Età: 28, Professione: Architetto”

Output: {"Nome":"Luca","Età":28,"Professione":"Architetto"}

Input: “- Titolo: Hamlet, Autore: Shakespeare, Anno: 1603”

Output: {"Titolo":"Hamlet","Autore":"Shakespeare","Anno":1603}

Ora converti:

Input: “- Prodotto: Tablet Z, Prezzo: 349.90, Disponibilità: Sì”

Output:

I risultati dell'esperimento che vengono raffigurati nella tabella sottostante, mostrano, come ci si potrebbe aspettare, che il modello più evoluto, che è anche il più costoso, ha performance migliori e riesce a generare testo più simile alla reference.

Model	BLEU	ROUGE-1	ROUGE-2	ROUGE-L
Gemini-2.5-pro	0.496	0.5982	0.4819	0.5973
Gemini-2.0-flash	0.4449	0.5643	0.4606	0.5643
Gemini-2.0-flash-lite	0.424	0.5562	0.4275	0.5534

Notiamo un aumento di 5.5 BLEU points di Gemini-2.5 pro rispetto al secondo modello più performante dove tradizionalmente un aumento 3 BLEU points sono considerati una variazione sostanziale . Per quanto riguarda invece la famiglia di parametri del Rouge score, su tutti i valori si possono notare performance migliori di Gemini-2.5-pro rispetto agli altri

modelli presi in esame. In particolare le differenze tra i valori Rouge-1 e Rouge-L per Gemini-2.5-pro e Gemini-2.5 Flash superano i 3 punti, ovvero la soglia per cui la differenza nell'output dei modelli è considerata apprezzabile.

In sintesi, le migliori prestazioni ottenute dal modello più costoso giustificano l'aumento di prezzo, soprattutto in questo contesto dove la qualità dell'output infrastrutturale è un requisito critico.

## **TEST SCALABILITA' E PERFORMANCE**

Per garantire che il sistema proposto sia in grado di gestire carichi variabili e rispondere in modo efficace sono stati condotti test di scalabilità e performance. L'obiettivo di questi test è valutare la capacità dell'architettura di mantenere livelli accettabili di latenza e throughput, nonché di assicurare la resilienza e la stabilità del sistema in condizioni operative differenti.

I test sono stati realizzati utilizzando K6, uno strumento open-source per il load testing, particolarmente adatto per simulare scenari realistici e carichi elevati su applicazioni web e API. Sono state applicate tre tipologie di test:

- **Scalability Test:** per verificare il comportamento del sistema al crescere progressivo del numero di utenti simultanei e identificare eventuali colli di bottiglia.
- **Spike Test:** per valutare la capacità dell'architettura di gestire improvvisi e rapidi picchi di traffico, simulando scenari reali come campagne promozionali o eventi inattesi.
- **Stress Test:** per determinare il punto di rottura del sistema e osservare come si comporta oltre i limiti operativi previsti, al fine di identificare le soglie massime di carico sostenibili.

Dopo una prima esecuzione di alcuni test è apparso subito evidente come il bottleneck principale fosse la chiamata all'API del large language model, essendo le performance di quest'ultimo fuori dal nostro controllo, influenzate da fattori come ad esempio il carico sul server che ospita il modello migliorabili semplicemente concordando con un fornitore un miglioramento del servizio oppure affittare GPU su cui ospitare un'istanza di un LLM dedicata al nostro servizio. Per meglio valutare le performance delle applicazioni abbiamo svolto sia test includendo l'attività di inferenza sia test escludendo l'attività di inferenza.

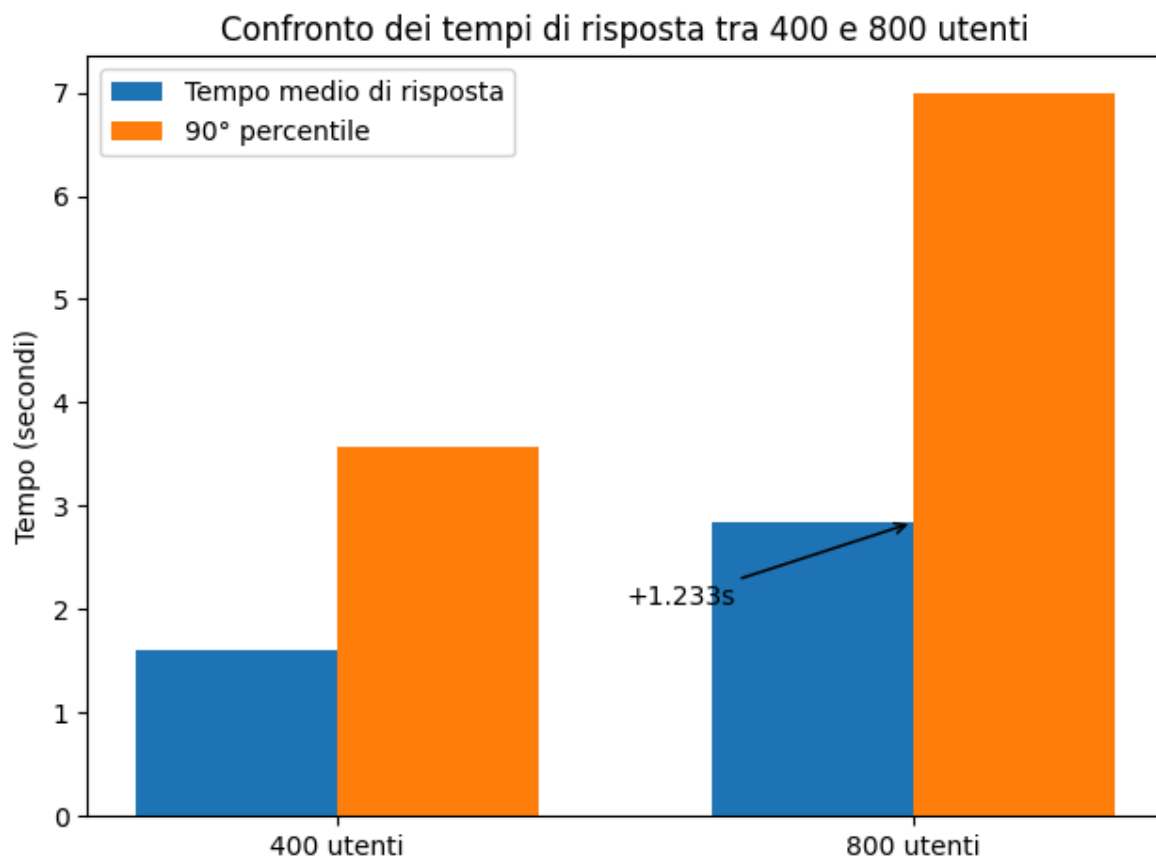
### **Scalability Test**

Il primo test effettuato è lo scalability test, si aumenta progressivamente il carico sul server con lo scopo di verificare come l'infrastruttura gestisce un aumento progressivo del carico di

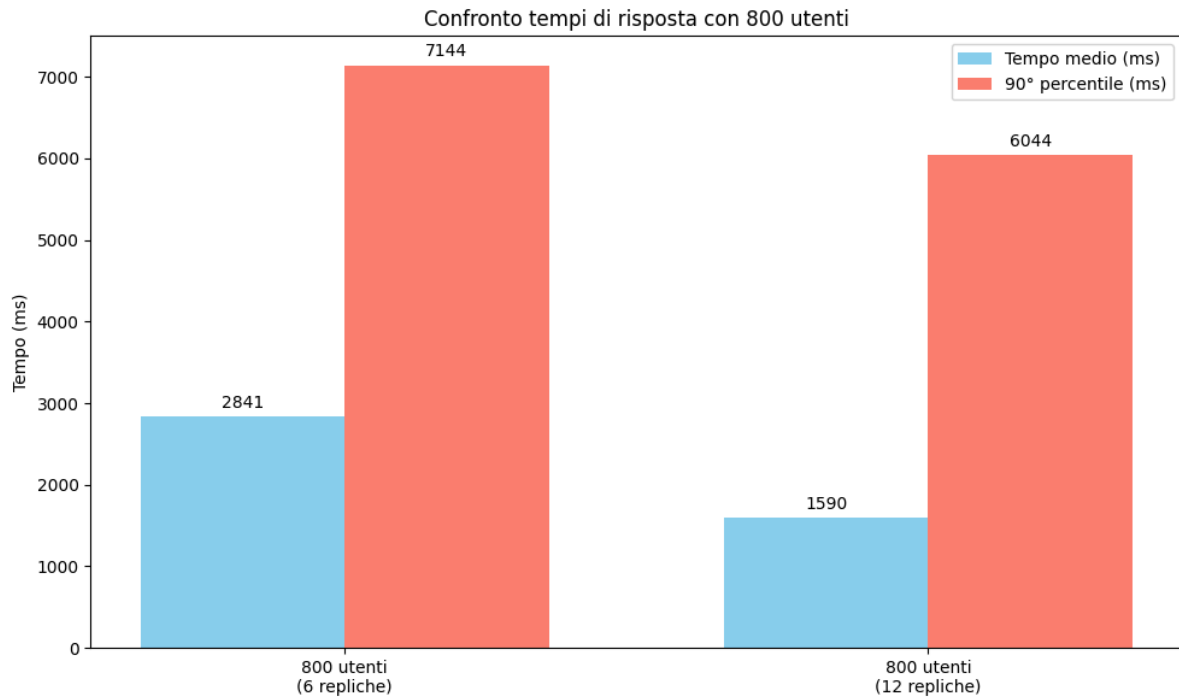
lavoro. Per quanto riguarda i test senza l'inferenza del modello abbiamo svolto 2 test, entrambi con un numero massimo di repliche del backend uguale a 6; nel primo test si raggiunge un picco di 400 utenti mentre nel secondo il picco di utenti è il doppio del test precedente ovvero 800 utenti.

Nel caso di 400 utenti non notiamo alcuna criticità particolare, le richieste hanno tutte esito positivo, il tempo medio di risposta è di 1,608 secondi con un minimo di 51 ms e un massimo di 8,16 secondi; con il 90% delle chiamate che riceve una risposta in meno di 3.577 secondi.

Considerando il caso con il picco di 800 utenti, la situazione cambia radicalmente; i tempi di risposta risultano estremamente dilatati, la risposta media è allungata di più di un secondo arrivando a 2.841, e il 90% delle chiamate ha una durata superiore ai 7 secondi. Per aumentare quindi i tempi di risposta del sistema risulta necessario aumentare il numero massimo di repliche del backend



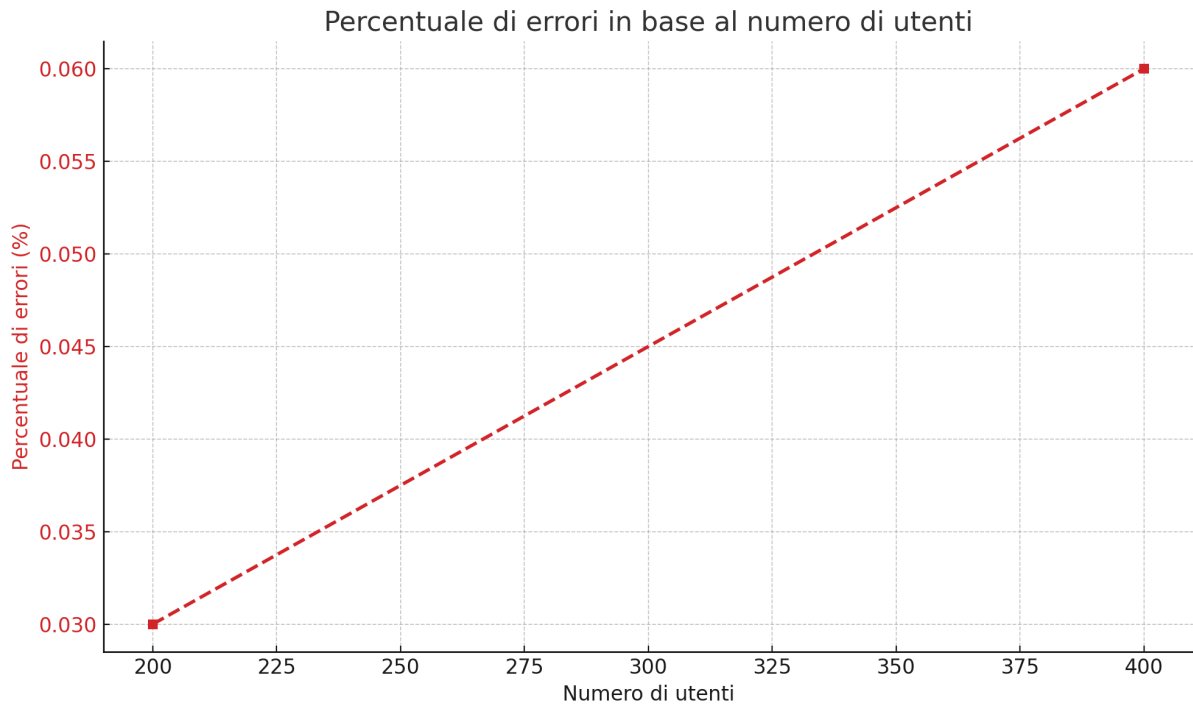
Abbiamo deciso quindi di effettuare il test di scalabilità da 800 utenti ma raddoppiando il numero di repliche per il backend per verificare la scalabilità dell'applicazione e la sua capacità ad utilizzare risorse hardware aggiuntive; da questi test è emerso che raddoppiando il numero di repliche il tempo di risposta medio scende a 1.590 ms diminuendo di circa il 45%. Diminuisce anche il novantesimo percentile, scendendo a 6.044 secondi.



Per quanto riguarda i test svolti con l'inferenza del modello, i numeri sono ovviamente molto più modesti ma danno una idea di quella che sono le performance reali del servizio e la qualità dell'esperienza utente media.

Sono stati svolti 2 test di scalabilità, uno con un picco di 200 utenti e l'altro con un picco di 400 utenti.

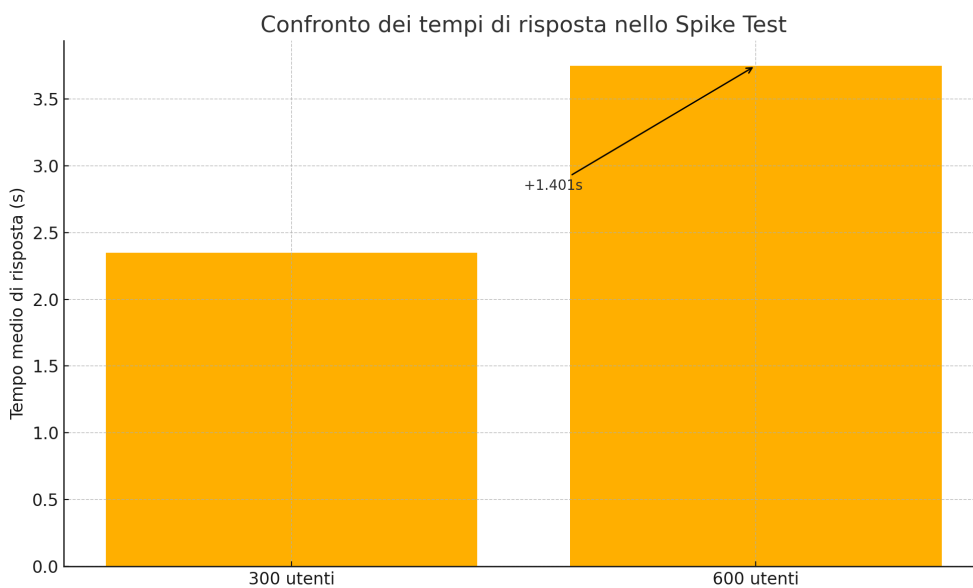
In entrambi i casi i tempi di risposta sono alti ma simili in quanto gran parte del tempo viene speso per l'inferenza del modello, la richiesta ha una durata media di 12 secondi, quello che cambia è la quantità di fallimenti, nel caso dei 200 utenti la percentuale di richieste con errore è lo 0.03%; questo numero aumenta all'aumentare degli utenti, con 400 utenti la percentuale di richieste che ricevono una risposta con un codice negativo raddoppia al 0.06%.



In questo caso aumentare le repliche avrebbe un effetto marginale o nullo sulle performance.

### Spike Test

Lo scopo dello Spike Test è verificare la tenuta delle performance del backend in seguito all'aumento improvviso di utenti. Per quanto riguarda i test svolti senza l'inferenza del modello sono stati testati 2 casi il primo con un picco di 300 utenti e il secondo con un picco di 600 utenti. Si è notato un aumento dei tempi di risposta passando da 2.348 secondi per il caso da 300 utenti a 3.749 nel caso da 600 utenti.



## **Stress Test**

Con lo stress test l'obiettivo è di valutare la capacità del sistema in condizioni particolarmente difficili, abbiamo scelto di testare l'applicazione con un picco di 600 utenti mantenendo un numero di repliche di 6 per il backend.

Durante lo stress test il sistema ha avuto un tempo medio di risposta di 18.048 secondi con il 95% delle richieste che hanno ottenuto una richiesta entro i 20.201 secondi.

Per svolgere la totalità dei test appena descritti sono stati spesi 219€ di cui 216€ solo di Vertex AI per l'utilizzo del modello.

## **COST ESTIMATION**

Il modulo di stima dei costi è progettato per fornire una valutazione automatizzata dei costi dell'infrastruttura cloud, partendo dallo stato effettivamente applicato su Google Cloud Platform (GCP).

A partire da un file Terraform generato tramite l'applicazione vengono estratte le risorse GCP a partire dal relativo file .tfstate. Per ogni risorsa identificata viene effettuato un mapping verso il corrispettivo servizio GCP.

Una volta determinati i servizi, il modulo interroga l'API Cloud Billing Catalog di GCP. Questa API espone l'elenco completo dei SKU (Stock Keeping Unit) e dei relativi prezzi aggiornati per ogni servizio. Per ciascuna risorsa, vengono applicati filtri dinamici basati sugli attributi specifici (es. tipo macchina, regione, storage size) al fine di isolare solo gli SKU rilevanti.

Infine, viene calcolato un costo totale sommando i prezzi unitari degli SKU filtrati, tenendo conto di unità di misura e granularità.

E' inoltre possibile consultare LLM per ricevere un feedback in linguaggi naturali riguardo ai costi del file di riferimento.

## **COSTI DI MANTENIMENTO**

Per quanto riguarda il mantenimento dell'applicazione è stato stimato un costo totale a partire dalle stime di spesa sui seguenti componenti:

<b>Service</b>	<b>Total Price (EUR)</b>
Backend (Cloud Run)	83.45
Frontend (Cloud Run)	40.94
Pub/Sub	60.30
Firestore	1.84
Cloud DNS	4.27
Cloud Load Balancing	16.95
IP Address	21.80
Data Transfer	59.81
Cloud Storage	85.39

**Totale stimato: €374.78**

Il prezzo per il mantenimento del prodotto si aggira intorno ai €374 al mese, questo dato si riferisce a un ammontare di 10 milioni di richieste al mese (all'incirca 3.7 req/sec per tutto il servizio).



## CONCLUSIONI

L'architettura progettata risulta robusta, scalabile ed economicamente sostenibile per scenari di utilizzo intensivo. L'unico punto critico identificato è legato alle performance del componente LLM, che rappresenta un elemento esterno alla nostra diretta gestione.

In termini di affidabilità, l'uso di servizi gestiti come Cloud Run, Pub/Sub e Firestore garantisce elevata disponibilità (fino al 99.9%) e tolleranza ai guasti, mentre la configurazione di load balancing e l'isolamento dei componenti e minimizzano il rischio di single points of failure.

I test di scalabilità hanno evidenziato come il sistema risponda positivamente all'aumento progressivo degli utenti evidenziando la configurazione dei meccanismi di autoscaling e la capacità dell'architettura di sfruttare risorse hardware secondo necessario.

L'adozione di un LLM dedicato su GPU o di SLA più alti col fornitore potrà ridurre la latenza e migliorare la stabilità e affidabilità del sistema sotto carichi elevati.

## BIBLIOGRAFIA

- <https://cloud.google.com/run/docs/overview/what-is-cloud-run?hl=it>
- <https://cloud.google.com/storage?hl=it#how-it-works>
- <https://grafana.com/docs/k6/latest/testing-guides/>
- <https://cloud.google.com/vertex-ai/generative-ai/pricing#token-based-pricing>
- <https://huggingface.co/datasets/autoiac-project/iac-eval>
- <https://arxiv.org/abs/2009.10297>
- <https://eurointervention.pcronline.com/article/the-syntax-score-on-its-way-out-or-towards-artificial-intelligence-part-ii>