LLM4SOC – False Positive Triage Assistant for IDS Alert Analysis

Federico Mancini, Samuele Mazziotti

07-2025

Indice

1 Introduzione

Nel contesto della sicurezza informatica, gli Intrusion Detection Systems (IDS) rappresentano strumenti fondamentali per il monitoraggio e l'identificazione di potenziali minacce nei confronti di reti e sistemi informativi. Tuttavia, una delle problematiche più comuni e gravose legate all'uso degli IDS è l'elevato numero di falsi positivi, ovvero segnalazioni di attività sospette che, in realtà, non rappresentano un pericolo concreto. Questo fenomeno comporta un notevole dispendio di tempo e risorse da parte degli analisti di sicurezza, i quali devono effettuare un triage manuale degli alert per identificare quelli realmente rilevanti.

Il progetto *LLM4SOC – False Positive Triage Assistant for IDS Alert Analysis* nasce con l'obiettivo di affrontare tale problema mediante l'uso di tecnologie di Intelligenza Artificiale, e in particolare dei Large Language Models (LLM). L'idea centrale è quella di costruire un assistente intelligente capace di:

- Analizzare automaticamente i log di alert generati da un IDS, con un focus specifico sui dataset AIT-ADS (Alert Interpretation Tool – Anomaly Detection System).
- Classificare ciascun alert come vero positivo o falso positivo.
- Fornire una spiegazione leggibile e coerente del motivo alla base della classificazione effettuata.

Negli ultimi anni, i LLM si sono dimostrati estremamente versatili in molteplici applicazioni, tra cui l'analisi del linguaggio naturale, la generazione di codice, la sintesi di documenti e – sempre più frequentemente – nel supporto alla cybersecurity. In particolare, nel contesto della sicurezza, i LLM stanno emergendo come strumenti utili per il supporto decisionale, la correlazione tra eventi di log, l'analisi automatica delle minacce e la generazione di spiegazioni per facilitare il lavoro degli analisti. Il loro punto di forza risiede nella capacità di comprendere il contesto semantico, correlare informazioni distribuite e produrre risposte in linguaggio naturale, rendendoli ideali per il triage di alert complessi.

Il sistema sarà interamente deployato su Google Cloud Platform (GCP), sfruttando strumenti infrastrutturali come Terraform per il provisioning automatizzato delle risorse, e sarà dotato di una dashboard web che permetterà agli utenti di caricare dataset, visualizzare e filtrare i risultati, esportarli ed interagire con il sistema tramite un'interfaccia chatbot.

2 Requisiti di sistema

L'obiettivo del progetto *LLM4SOC* è stato quello di costruire una piattaforma efficiente e affidabile per l'analisi e la classificazione automatica degli alert provenienti da un Intrusion Detection System, in particolare a partire da dataset di tipo AIT-ADS. Questi dataset contengono log strutturati generati da un IDS in formato CSV, JSON o JSONL includendo diversi campi (timestamp, indirizzi IP, porta, protocollo, ecc.).

Dal punto di vista funzionale, il sistema consente all'utente di caricare file con estensione .csv, .json o .jsonl tramite un'apposita interfaccia web. Una volta caricato il file, i dati vengono inviati al server per poi essere analizzati e, attraverso determinate API (`/upload-dataset` e `/analyze-dataset`), si effettua il parsing e la classificazione automatica degli alert mediante un LLM. Ogni alert viene etichettato come *real threat*, *false positive* o *undetermined* (stato in cui il modello, per ambiguità o mancanza di contesto, non riesce a determinare con certezza la natura dell'alert), con l'aggiunta di una spiegazione testuale che giustifica la classificazione ricevuta.

Tutti i risultati dell'analisi vengono visualizzati in una dashboard interattiva che presenta gli alert classificati in forma tabellare, con opzioni di filtro, ricerca e interazione. L'utente può esplorare i dati, richiedere chiarimenti tramite un chatbot integrato e scaricare i risultati in formato JSON o CSV.

Il sistema è stato pensato per essere scalabile, grazie al deploy su Google Cloud Platform e all'uso di Terraform per la gestione automatizzata dell'infrastruttura. Questo approccio consente non solo il provisioning delle risorse cloud, ma anche una facile replicazione, estensione e manutenzione dell'intero sistema.

3 Architettura del sistema

Il sistema *LLM4SOC* è stato progettato secondo un'architettura modulare e scalabile, che separa le responsabilità tra frontend e backend, garantendo al contempo un'elevata integrazione tra le componenti.

Il frontend, sviluppato in React, costituisce l'interfaccia utente principale. Consente di caricare dataset, visualizzare e filtrare i risultati dell'analisi, interagire con un assistente basato su LLM e scaricare i dati elaborati. Ogni funzionalità è pensata per semplificare il lavoro degli analisti SOC (Security Operation Center), offrendo un'interazione fluida anche su dataset di grandi dimensioni.

Il backend è composto da più servizi distribuiti su Google Cloud Platform, ciascuno responsabile di una fase specifica del processo: gestione delle API, orchestrazione delle analisi, invocazione del modello linguistico per la classificazione degli alert, aggregazione e consolidamento dei risultati. L'uso di componenti serverless e di strumenti come Cloud Run, Cloud Tasks e Vertex AI ha permesso di ottenere un sistema flessibile, reattivo e facilmente scalabile.

L'intera infrastruttura è definita tramite Terraform, seguendo un approccio Infrastructure as Code. Questo consente il provisioning automatizzato delle risorse cloud e facilita il mantenimento e l'evoluzione del sistema nel tempo.

3.1 Componenti Frontend

Il frontend dell'applicazione *LLM4SOC* ha l'obiettivo di offrire un'interfaccia utente semplice, interattiva e funzionale per la gestione dell'intero flusso di analisi degli alert generati da un sistema di Intrusion Detection (IDS). Il suo ruolo è quello di collegare l'utente finale con la logica applicativa implementata lato backend, permettendo:

- Il caricamento dei dataset contenenti log di alert;
- La visualizzazione e l'analisi dei risultati prodotti dal modello LLM;
- L'interazione diretta con un chatbot, attraverso richieste di chiarimento e spiegazioni personalizzate in merito a singoli alert;
- L'esportazione dei dati analizzati in formato strutturato.

L'interfaccia è stata progettata per supportare il lavoro di analisti SOC e semplificare il processo di triage degli alert, minimizzando il tempo necessario per distinguere i falsi positivi da quelli reali. Il frontend si integra con il backend attraverso richieste HTTP asincrone, comunicando con endpoint REST esposti da un server FastAPI. Attraverso questa comunicazione, l'interfaccia è in grado di gestire l'intero ciclo di vita dei dati: dall'upload del file, all'attesa dei risultati dell'analisi, fino alla presentazione finale e al dialogo testuale con LLM.

Per soddisfare tali obiettivi, è stato scelto React come framework di sviluppo, poiché consente una progettazione modulare, una gestione efficace dello stato e una semplice integrazione con sistemi esterni tramite API REST. Il frontend è organizzato secondo un'architettura a componenti, in cui ciascun modulo ha responsabilità specifiche e ben isolate. La comunicazione tra questi avviene attraverso meccanismi strutturati come le properties, utilizzate per passare dati e configurazioni, e lo stato locale, usato per gestire informazioni dinamiche interne al singolo componente.

Il punto d'ingresso principale è il file `App.js`, che definisce la struttura complessiva della pagina e funge da orchestratore tra i vari componenti. In `App.js` vengono gestiti gli stati globali dell'interfaccia, come il caricamento dei dati, la lista degli alert analizzati, gli alert selezionato per la chatbot e altri ancora.

A partire da `App.js`, l'applicazione è suddivisa nei seguenti componenti principali:

- `UploadFile.js`: consente all'utente di caricare un file locale e avvia l'intero flusso di analisi. Internamente gestisce le chiamate asincrone all'API per il caricamento, l'analisi e il recupero dei risultati, con funzionalità di retry automatico e timer di attesa.

- `AlertTable.js`: visualizza i risultati dell'analisi in forma tabellare, con supporto a filtri per classificazione, ricerca testuale e selezione di uno o più alert. Utilizza una strategia di rendering virtualizzato per mantenere le prestazioni elevate anche in presenza di dataset molto grandi.
- `Chatbot.js`: gestisce l'interazione testuale tra l'utente e il sistema, permettendo di inviare domande su uno o più alert selezionati e ricevere spiegazioni generate dal LLM. L'interfaccia prevede la visualizzazione dei messaggi scambiati, lo stato di caricamento e la gestione degli errori.
- `apiService.js`: contiene le funzioni per la comunicazione con il backend tramite fetch API, dove ogni endpoint è incapsulato in una funzione specifica.

Nel complesso, la struttura dell'applicazione React è pensata per massimizzare modularità, riusabilità e chiarezza del codice, facilitando eventuali estensioni future o sostituzioni dei componenti esistenti.

3.1.1 Gestione del caricamento e analisi dei file

Il caricamento e l'analisi dei file costituiscono la fase iniziale del flusso operativo del sistema. Queste funzionalità sono gestite principalmente all'interno del componente `UploadFile.js`, il quale fornisce all'utente un'interfaccia per selezionare un file locale e avviare il processo di analisi automatica.

Una volta che il file è stato selezionato, il componente richiama la funzione *uploadFileToAPI* (definita in `apiService.js`), che costruisce un oggetto FormData e lo invia al backend tramite una richiesta POST all'endpoint `/upload-dataset`. Se il caricamento va a buon fine, viene attivata la successiva richiesta GET a `/analyze-dataset`, tramite la funzione *analyzeAlertsOnServer*, che avvia l'analisi degli alert da parte del backend. Poiché questa elaborazione può richiedere tempo, il sistema utilizza un approccio asincrono basato su polling, verificando periodicamente lo stato dell'elaborazione. In particolare, viene attivato un ciclo temporizzato che invia ripetutamente richieste GET all'endpoint `/result`, tramite la funzione *fetchResultsFromAPI*, per verificare se l'analisi è stata completata. Il ciclo termina quando i risultati vengono restituiti con successo o quando l'utente interrompe manualmente l'esecuzione tramite l'apposito pulsante di stop.

Durante questa fase, lo stato dell'interfaccia viene aggiornato per mostrare in tempo reale il tempo trascorso, il numero di tentativi e un messaggio di caricamento. In caso di errore, viene mostrato un messaggio apposito che lo descrive nella parte superiore dell'interfaccia ed è possibile ripetere l'operazione.

3.1.2 Visualizzazione e interazione con la tabella alert

La visualizzazione dei risultati analizzati dal backend è affidata al componente `AlertTable.js`, che rappresenta uno degli elementi centrali dell'interfaccia utente. Questo componente riceve in input l'elenco completo degli alert analizzati, ottenuti tramite polling asincrono successivamente al caricamento del file, e fornisce una rappresentazione tabellare interattiva dei dati.

La tabella visualizzata nel frontend è strutturata in modo tale che ogni riga corrisponde a un singolo alert, mentre le colonne rappresentano i principali attributi informativi, tra cui: l'identificativo univoco dell'alert, il timestamp convertito in un formato leggibile, la classificazione assegnata (falso positivo, minaccia reale o indeterminato) e la spiegazione testuale generata dal LLM.
L'interfaccia utente consente di filtrare gli alert in base alla classe di rischio e all'intervallo temporale, ed è inoltre dotata di una funzionalità di ricerca testuale globale. Quest'ultima permette di individuare rapidamente specifici alert attraverso parole chiave contenute nella spiegazione o negli identificativi.

Dal punto di vista dell'utente, la tabella mantiene un'elevata fluidità e reattività anche durante lo scorrimento di migliaia di righe, senza rallentamenti o interruzioni percettibili. L'interfaccia risulta stabile e immediata nell'interazione, garantendo una navigazione continua e senza ritardi, indipendentemente dalla dimensione del dataset visualizzato.

Il componente gestisce anche la selezione degli alert: l'utente può cliccare su una riga per selezionare un singolo alert o utilizzare una casella di controllo per selezioni multiple. Questa funzionalità è strettamente integrata con il componente `Chatbot.js`, poiché la selezione degli alert viene propagata tramite properties per consentire al chatbot di fornire risposte contestualizzate ai log selezionati.

Infine, la tabella include una funzione di esportazione, che consente all'utente di scaricare i risultati filtrati in formato CSV o JSON. Questa funzionalità è implementata direttamente all'interno di `AlertTable.js`, sfruttando le API browser-native per la generazione dinamica di file e il trigger del download. Questo viene fatto in modo da permettere all'utente di salvare localmente un gruppo specifico di alert determinato mediante i filtri applicati nella visualizzazione.

3.1.3 Chatbot e comunicazione con LLM

Una delle funzionalità distintive dell'interfaccia è l'integrazione di un chatbot, progettato per facilitare l'interazione naturale tra l'utente e il sistema di analisi. Questo modulo, implementato nel file `Chatbot.js`, consente all'utente di porre domande o richiedere chiarimenti su specifici alert classificati dal modello LLM.

La logica del componente prevede che l'utente selezioni uno o più alert dalla tabella, i cui dati vengono passati al chatbot come properties. L'interfaccia include un campo di input testuale dove l'utente può digitare la propria domanda e un pulsante per inviare la richiesta. Una volta attivata l'interazione, il messaggio dell'utente e gli alert selezionati vengono incapsulati in un oggetto JSON e inviati al backend tramite la funzione *sendMessageToChatAPI*, definita in `apiService.js`.

L'endpoint REST coinvolto in questa comunicazione è `/chat`, il quale si occupa di orchestrare la costruzione del prompt, l'interrogazione del LLM e la restituzione della risposta in linguaggio naturale. Gestisce anche eventuali errori di rete o timeout, mostrando all'utente messaggi adeguati e mantenendo lo stato coerente.

Una volta ricevuta la risposta dal backend, questa viene visualizzata in un'area di conversazione all'interno del componente, organizzata secondo il classico paradigma chat-based: messaggi dell'utente a destra, risposte del sistema a sinistra. La struttura dell'interfaccia è pensata per supportare scambi multipli e sequenziali, mantenendo lo storico della conversazione finché la sessione resta attiva.

3.1.4 Gestione dello stato e comunicazione tra componenti

La gestione dello stato all'interno dell'applicazione React è centralizzata nel componente `App.js`. L'approccio adottato segue un modello top-down, in cui lo stato globale è definito nel livello superiore e propagato ai componenti figli tramite properties.

Nel dettaglio, all'interno di `App.js` sono dichiarate variabili di stato tramite l'hook *useState*, tra cui:

- *alerts*: contiene l'elenco completo degli alert analizzati.
- *selectedAlert* e *selectedAlerts*: usati per gestire l'interazione tra tabella e chatbot.
- *uploadStatus*, *error*, *elapsedTime*, *fetchAttempts*: gestiscono lo stato della richiesta asincrona e il feedback utente durante l'analisi.

Il componente `UploadFile.js` riceve le funzioni di aggiornamento dello stato (es. *setAlerts*, *setUploadStatus*) come properties, così da poter modificare direttamente lo stato centrale durante il caricamento, l'analisi e il polling dei risultati. In modo analogo, `AlertTable.js` riceve lo stato degli alert e le funzioni per aggiornare gli alert selezionati, permettendo la sincronizzazione tra selezione in tabella e uso nel chatbot.

Il componente `Chatbot.js` accede a *selectedAlerts* e aggiorna il proprio stato interno per gestire il messaggio dell'utente, lo storico della conversazione e la risposta ricevuta. Tuttavia, non modifica direttamente lo stato globale, mantenendo così un'architettura reattiva e ben separata tra livelli di responsabilità.

3.1.5 Chiamate API e modularizzazione della logica di rete

La comunicazione tra il frontend e il backend è gestita attraverso chiamate HTTP asincrone incapsulate in un modulo dedicato: `apiService.js`. Questo file svolge un ruolo cruciale nel separare la logica di rete dalla logica di presentazione, migliorando la leggibilità, la riusabilità del codice e la manutenibilità dell'applicazione.

Il file `apiService.js` definisce ed esporta una collezione di funzioni asincrone, ciascuna delle quali incapsula la logica per interagire con un endpoint specifico esposto dal backend. Le principali funzioni implementate sono le seguenti:

- *uploadFileToAPI()*: invia un file selezionato dall'utente al backend tramite una richiesta POST verso l'endpoint `/upload-dataset`. Il file viene incapsulato in un oggetto FormData, che consente il corretto invio multipart/form-data, necessario per trasmettere file binari e dati testuali nella stessa richiesta HTTP.
- *analyzeAlertsOnServer()*: invia una richiesta GET all'endpoint `/analyze-dataset` per avviare il processo di analisi automatica del dataset caricato.
- *fetchResultsFromAPI()*: invia richieste GET all'endpoint `/result`, recuperando i risultati dell'analisi. Questa funzione viene utilizzata nel ciclo di polling implementato in `UploadFile.js`, che monitora lo stato dell'analisi fino alla sua conclusione o al raggiungimento del numero massimo di tentativi.
- *sendMessageToChatAPI()*: invia al backend un messaggio di testo e una lista di alert selezionati, tramite richiesta POST all'endpoint `/chat`. La risposta, generata dal LLM, viene poi mostrata nella chat.

Tutte le funzioni sono implementate tramite il paradigma *async/await*, e integrano un controllo esplicito sullo stato della risposta HTTP. In presenza di errori, viene sollevata un'eccezione che può essere intercettata dal componente chiamante per gestire correttamente lo stato dell'interfaccia utente.

L'utilizzo di `apiService.js` come modulo di servizio centralizzato per le chiamate di rete consente di migliorare la manutenibilità del codice e di sostituire agevolmente gli endpoint durante le fasi di sviluppo o deploy. Inoltre, questa soluzione previene la duplicazione della logica di rete e uniforma la gestione degli errori e delle intestazioni HTTP.

3.1.6 Scelte di UX/UI e stile

L'interfaccia utente dell'applicazione *LLM4SOC* è stata progettata per garantire elevata usabilità, chiarezza visiva e reattività, in coerenza con il contesto d'uso: l'analisi di alert generati da un IDS, spesso numerosi e complessi, da parte di utenti con profilo tecnico. Le scelte progettuali adottate si riflettono sia nella struttura grafica che nella dinamica di interazione con l'utente.

Lo stile visivo dell'interfaccia è definito nel file `App.css`. Il contrasto elevato tra sfondo e testo favorisce la leggibilità, mentre l'uso di una codifica cromatica (verde per operazioni completate con successo, rosso per errori, blu per processi in corso) fornisce un feedback immediato sullo stato delle attività.

Dal punto di vista dell'esperienza utente, ogni componente dell'interfaccia restituisce un feedback visivo chiaro e contestuale. Durante il caricamento dei file, l'interfaccia visualizza informazioni relative al tempo trascorso e al numero di tentativi di elaborazione effettuati. In caso di errore, viene mostrato un messaggio informativo nella parte superiore dello schermo.

L'interazione con la tabella degli alert è stata ottimizzata per garantire fluidità anche in presenza di dataset di grandi dimensioni. La selezione degli alert può avvenire tramite clic singolo o multiplo, e viene riflessa in tempo reale nell'attivazione del modulo di interazione conversazionale. La visualizzazione privilegia la concentrazione sull'informazione, evitando animazioni superflue o elementi grafici distraenti.

Il modulo chatbot è integrato nella parte superiore dell'interfaccia ed è accessibile esclusivamente quando un alert risulta selezionato. Questa scelta progettuale previene interazioni non contestuali, guidando l'utente in un flusso di analisi coerente. I messaggi generati durante l'interazione sono presentati secondo uno schema conversazionale, con alternanza visiva tra le richieste dell'utente e le risposte fornite dal sistema, differenziate tramite colori e allineamento.

3.1.7 Performance

Uno degli aspetti più delicati nello sviluppo del frontend è stato garantire un'interfaccia reattiva anche in presenza di dataset di grandi dimensioni, come quelli presenti nel contesto IDS. La gestione efficiente delle performance è stata affrontata su più livelli: rendering, caricamento dati e aggiornamento dell'interfaccia.

La sfida principale è stata garantire una visualizzazione tabellare fluida e reattiva anche in presenza di dataset di grandi dimensioni (fino a diverse migliaia di righe). Per mitigare i problemi legati al caricamento completo del DOM (Document Object Model), è stata adottata la libreria *react-window*, specializzata nel rendering virtualizzato. Questa soluzione consente di generare dinamicamente solo le righe attualmente visibili nella finestra dell'utente, evitando di renderizzare l'intero dataset contemporaneamente. A differenza di un approccio tradizionale basato su *.map()*, che crea un componente React per ogni riga, *react-window* riduce significativamente l'uso della memoria e della CPU, garantendo prestazioni costanti anche con volumi elevati di dati.
L'adozione di questa tecnica ha migliorato in modo tangibile la reattività dell'interfaccia, contribuendo a un'esperienza utente più stabile e scalabile, in linea con le esigenze di utilizzo in scenari reali.

Durante la fase di caricamento e analisi del file invece, il sistema utilizza un approccio asincrono basato su polling progressivo. Il componente `UploadFile.js` invia richieste periodiche all'endpoint di analisi tramite la funzione *fetchResultsFromAPI*, aggiornando dinamicamente l'interfaccia in base allo stato corrente del backend. Questo meccanismo consente di informare in tempo reale l'utente sull'avanzamento dell'elaborazione, evitando attese non informative o blocchi dell'interfaccia.

Il tempo trascorso dall'inizio del polling viene misurato tramite un timer basato su *setInterval*, mentre un contatore dei tentativi (*fetchAttempts*) viene incrementato a ogni richiesta. Entrambe le variabili (*elapsedTime* e *fetchAttempts*) sono gestite tramite gli hook *useState* e *useEffect*, con logiche di reset automatico al termine del processo. La visualizzazione di queste metriche rende trasparente il comportamento del sistema all'utente finale.

L'interfaccia è stata progettata per gestire in modo controllato eventuali errori temporanei durante le chiamate al backend. In caso di risposta HTTP non valida, il polling non viene interrotto bruscamente: il sistema effettua un nuovo tentativo automaticamente e mostra un messaggio informativo all'utente. È inoltre prevista la possibilità di ripetere manualmente il caricamento in caso di errore persistente.
Questa strategia di degradazione controllata consente di evitare crash o blocchi improvvisi, contribuendo alla resilienza complessiva dell'interfaccia utente.

3.2 Componenti Backend

3.2.1 Infrastruttura GCP

L'infrastruttura del sistema è interamente ospitata su Google Cloud Platform (GCP), sfruttando in modo integrato diversi servizi gestiti per garantire scalabilità, modularità e semplicità di deployment. I principali componenti dell'architettura sono containerizzati tramite Docker e definiti attraverso manifesti YAML, Dockerfile personalizzati e file `requirements.txt` per la gestione delle dipendenze. L'intero ciclo di build e deploy è automatizzato tramite Cloud Build, configurato come sistema di CI/CD (Continuous Integration/Continuous Deployment). Ogni modifica al codice sorgente, archiviato in una repository GitHub, genera automaticamente una nuova immagine Docker che viene verificata e pubblicata su Cloud Run.

Tutti i file intermedi e finali, inclusi i dataset caricati, i risultati classificati, le metriche e i file di configurazione condivisi, sono archiviati nel servizio Cloud Storage. Questo garantisce l'accesso condiviso e persistente a ogni componente del sistema, indipendentemente dalla loro collocazione logica o istanza di esecuzione.

Il modulo principale di elaborazione, detto *worker*, è deployato come servizio containerizzato su Cloud Run, piattaforma serverless che consente l'esecuzione di container in risposta a richieste HTTP. Per realizzare un'elaborazione distribuita ed efficiente dei dataset, viene utilizzato Cloud Tasks, che consente di orchestrare migliaia di richieste asincrone verso il worker, adattando dinamicamente la frequenza delle invocazioni in base alla capacità computazionale disponibile.

Ogni richiesta al worker innesca a sua volta una serie di chiamate parallele verso il LLM ospitato su Vertex AI, la piattaforma d'intelligenza artificiale di Google. Vertex AI permette l'uso di modelli avanzati come Gemini, offrendo un'integrazione scalabile e gestita, ideale per scenari con carichi variabili e requisiti elevati di affidabilità.

Una volta terminata l'elaborazione di tutti i batch, viene attivato il *merge handler*, un componente containerizzato deployato anch'esso su Cloud Run. Il suo compito è aggregare i file parziali prodotti durante la classificazione e generare output unificati.

Il merge handler è attivato in modo automatico tramite Cloud Eventarc, servizio fully managed che consente di definire trigger reattivi su eventi GCP, come la creazione di nuovi file nei bucket del Cloud Storage. Gli eventi vengono consegnati tramite Pub/Sub, mentre l'invocazione dei componenti è affidata a un service account configurato con i permessi minimi necessari.

Infine, il controllo degli accessi e la sicurezza dell'infrastruttura sono gestiti tramite il sistema IAM (Identity and Access Management). Sono stati definiti ruoli granulari e applicata una politica di least privilege, assegnando a ciascun servizio soltanto i permessi strettamente indispensabili per le proprie operazioni.

3.2.2 Terraform

L'intera infrastruttura è stata definita tramite Terraform. La configurazione è organizzata in più file specializzati, ciascuno responsabile della definizione e gestione di una specifica componente su Google Cloud Platform.

Il file `vm.tf` definisce una macchina virtuale Compute Engine di tipo *e2-medium*, configurata per ospitare i server FastAPI. L'istanza viene inizializzata con immagine base *Debian 11* e prevede uno script di startup che installa automaticamente Python, le dipendenze del progetto e clona la repository GitHub contenente il backend. Il service account associato alla VM, generato automaticamente all'abilitazione del servizio Compute Engine, è dotato dei permessi minimi indispensabili per interagire con i servizi GCP utilizzati dal sistema, come Vertex AI, Cloud Storage e Cloud Tasks. La separazione dei privilegi è ulteriormente affinata nel file `iam.tf`, dove sono specificati i binding IAM relativi a ciascun account.

Per rendere accessibile il backend, il file `firewall.tf` configura una regola in ingresso sulle porte 8000 e 8001, abilitando il traffico HTTP verso le istanze da fonti esterne, con possibilità di filtraggio su base IP e tag di rete.

La componente di elaborazione asincrona è gestita da un servizio containerizzato su Cloud Run, descritto nel file `cloud_run_worker.tf`. Questo modulo specifica l'immagine Docker da utilizzare, i limiti di memoria, il timeout e il service account con cui il servizio viene eseguito. L'invocazione è protetta da autenticazione e i permessi sono assegnati in modo restrittivo per garantire l'accesso solo da parte di componenti autorizzati. Il componente responsabile dell'aggregazione dei risultati, il merge handler, è definito nel file `cloud_run_function_merge_handler.tf`, anch'esso configurato come servizio Cloud Run containerizzato, con impostazioni analoghe in termini di policy di sicurezza e risorse assegnate.

Per abilitare l'attivazione automatica del merge handler al termine dell'elaborazione dei batch, è stato configurato un trigger Eventarc nel file `triggers.tf`. Il trigger è sensibile a eventi di tipo `google.cloud.storage.object.v1.finalized` provenienti dal bucket condiviso e invoca il servizio di merge ogni volta che viene creato o modificato un file.

Tutti i parametri configurabili dell'infrastruttura, inclusi nome del progetto, ID bucket, zona, nomi dei servizi e variabili ambientali, sono centralizzati nei file `variables.tf` e `terraform.tfvars`. Le versioni di Terraform e dei provider GCP utilizzati sono esplicitate in `versions.tf`.

3.2.3 Server FastAPI

Il sistema include due istanze distinte di server FastAPI: il *server principale*, che espone l'interfaccia RESTful per l'utente e coordina il processo di analisi degli alert; il *server secondario*, dedicato esclusivamente alla gestione dei benchmark. Quest'ultimo viene eseguito su una porta separata per evitare interferenze e condizioni di deadlock.

Il server principale rappresenta il punto d'ingresso per tutte le operazioni interattive eseguite dall'utente. Esso espone una serie di endpoint (mediante metodi GET e POST) che consentono l'interazione diretta con il sistema. Le principali funzionalità comprendono il caricamento dei dataset (`/upload-dataset`), l'avvio dell'analisi (`/analyze-dataset`), la consultazione dei risultati (`/result`) e la comunicazione con il sistema chatbot (`/chat`). Sono inoltre previsti endpoint per il monitoraggio e la gestione del sistema, come `/health` e `/batch-result-status`.

Per iniziare l'analisi, l'utente carica un file con estensione .json, .jsonl o .csv nel bucket di sistema, inviando una richiesta POST all'endpoint `/upload-dataset`. Il server si occupa di completare l'upload e di generare i metadati associati, tra cui: nome del dataset (dataset_name), percorso nel bucket (dataset_path), numero di righe (num_rows), numero di colonne (num_columns), lista dei campi (features) e tipologia di contenuto (content_type). I dati e i relativi metadati sono salvati nella directory `datasets/` del bucket.

L'analisi viene avviata inviando una richiesta POST all'endpoint `/analyze-dataset`, specificando il nome del dataset. Il server esegue innanzitutto una pulizia delle directory `batch_results/` e `batch_metrics/` per assicurare che i file generati siano relativi esclusivamente all'elaborazione corrente. Successivamente viene rimosso il file `merge_lock.flag`, che funge da meccanismo di lock per il processo di merge, impedendo condizioni di concorrenza non sicure che possano portare a una sovrascrittura dei dati e/o alla loro perdita. La sua eliminazione consente al componente merge handler di monitorare nuovamente la directory `batch_results/` e attivare il processo di aggregazione dei risultati una volta completati tutti i batch.

Il server scarica i metadati dal bucket e aggiorna i parametri num_batches e batch_size, che possono essere stati modificati in seguito all'esecuzione di un benchmark. I metadati aggiornati vengono quindi salvati nuovamente nel bucket e passati alla funzione enqueue_batch_analysis_tasks, la quale si occupa di instanziare richieste concorrenti per ciascun batch. Ogni richiesta include: l'identificativo del batch (batch_id), gli indici degli alert che lo delimitano (start_row, end_row), la sua dimensione, il nome del dataset e il relativo percorso.
L'endpoint coinvolto restituisce un esito che conferma esclusivamente l'avvenuto avvio dell'elaborazione asincrona, fornendo al contempo una versione aggiornata dei metadati. Questa scelta progettuale nasce dall'esigenza di gestire in modo efficiente carichi computazionali elevati: infatti, nonostante l'impiego di strategie volte a migliorare prestazioni e scalabilità, l'analisi di dataset costituiti da decine di migliaia di righe resta un'operazione intensiva, il cui completamento può richiedere diversi minuti.
Per consentire il monitoraggio dello stato di avanzamento, è stato predisposto l'endpoint /batch-results-status, il quale restituisce informazioni aggiornate sull'elaborazione, tra cui lo stato corrente (pending, partial o completed), il numero di batch completati rispetto al totale e il nome del dataset in analisi.

Una volta completata l'elaborazione da parte del worker e del successivo processo di merge, i risultati e le metriche aggregate vengono resi disponibili all'interno delle directory `results/` (in formato JSON) e `metrics/` (in formato JSON e CSV). Poiché tali directory raccolgono i dati relativi a tutte le analisi eseguite nel tempo, l'utente può accedere ai risultati specifici di un dataset attraverso una richiesta all'endpoint `/result`, indicando esplicitamente il nome del dataset di riferimento.

L'endpoint `/chat` riceve da parte dell'utente le richieste contenenti nel payload uno o più alert e una domanda o un commento riferito agli stessi. Queste sono poi inoltrate al worker, il quale sfrutta le capacità del modello LLM per generare una risposta e restituirla al frontend, completando il ciclo di interazione.

Per l'esecuzione dei benchmark si è reso necessario introdurre un secondo server FastAPI, separato dal backend principale. Questa scelta architetturale nasce dall'esigenza di evitare condizioni di deadlock causate da chiamate HTTP effettuate dal processo stesso del server verso i propri endpoint.
In fase di benchmark, viene simulato un elevato numero di richieste indirizzate al backend; se tali richieste venissero gestite dallo stesso processo, il server resterebbe bloccato in attesa di una risposta che non potrà mai essere fornita, generando una situazione di stallo. Per ovviare a tale problematica, gli endpoint specifici per la gestione del benchmark sono stati implementati in uno script autonomo, denominato `benchmark.py`, avviato su una porta distinta rispetto a quella del server principale (`app.py`). I due server, pur essendo separati a livello di processo, condividono lo stesso spazio di progetto, il che consente loro di accedere a moduli e risorse comuni.

Gli endpoint di riferimento sono:

- `/start-benchmark`: consente l'avvio del benchmark sul dataset selezionato, configurando automaticamente i parametri di test.
- `/stop-benchmark`: permette la terminazione forzata dell'esecuzione in corso.
- `/benchmark-status`: fornisce lo stato corrente del processo.

3.2.4 Worker

Il componente *worker*, implementato come servizio su Google Cloud Run, consiste nel server incaricato esclusivamente dell'elaborazione computazionale, in risposta alle richieste ricevute dal server FastAPI principale. Il suo ruolo è quello di processare i dati relativi agli alert, sia in fase di analisi massiva dei dataset sia nel trattamento di richieste in linguaggio naturale ricevute tramite chatbot. Il worker non espone funzionalità all'utente finale, ma opera come backend interno nell'architettura del sistema.

Una volta caricato nel bucket, il dataset viene suddiviso logicamente in batch e per ciascuno il server principale genera una richiesta HTTP indirizzata al worker, che viene registrata come task all'interno del servizio Cloud Tasks. Questi task vengono poi eseguiti in parallelo, consentendo l'elaborazione concorrente dei batch da parte del worker, dove ogni richiesta contiene i metadati necessari all'elaborazione del batch.

Ricevuta la richiesta, il worker a sua volta elabora il batch sfruttando meccanismi di concorrenza interna: tramite il modulo `asyncio`, invia chiamate asincrone a Gemini per analizzare simultaneamente i singoli alert contenuti nel batch. I risultati dell'elaborazione e le relative metriche vengono infine salvati in formato strutturato rispettivamente all'interno delle directory `batch-results/` e `batch-metrics/` del bucket remoto.

Gli endpoint che il worker espone sono i seguenti:

- `/health`: utilizzato per il monitoraggio della disponibilità del servizio da parte del server principale.
- `/reload-config`: permette il ricaricamento delle variabili di configurazione gestite dal *Resource Manager*, un modulo singleton responsabile dell'inizializzazione dei parametri ambientali al momento del bootstrap del servizio.
- `/run-batch`: riceve una richiesta contenente le informazioni del batch da analizzare, estrae i dati dal file sorgente, interroga il modello Gemini per ciascun alert, calcola le metriche di performance e salva i risultati nel bucket remoto.
- `/run-chatbot`: riceve il messaggio dell'utente in linguaggio naturale, insieme a uno o più alert selezionati, ed effettua una richiesta al modello Gemini per ottenere una risposta da visualizzare nella chat della dashboard.

3.2.5 Merge handler

Il *merge handler* è un componente serverless eseguito come funzione su Google Cloud Run, progettato per aggregare automaticamente i risultati e le metriche generati durante l'analisi di un dataset. A differenza del worker, implementato come servizio container persistente, il merge handler è stato implementato come Cloud Run Function per sfruttare la possibilità di essere invocato automaticamente tramite trigger al verificarsi di eventi sul bucket di sistema.

La funzione monitora costantemente le directory `batch-results/` e `batch-metrics/`, e viene attivata automaticamente ogni volta che un file viene aggiunto o modificato all'interno del bucket, in seguito alla configurazione di un trigger sull'evento `google.cloud.storage.object.v1.finalized`. Tuttavia, a causa delle limitazioni dell'API GCP nella definizione di filtri granulari sui percorsi osservati, il trigger può scattare anche in risposta a modifiche avvenute in directory non pertinenti.
Per mitigare questo comportamento, sono stati implementati a livello applicativo controlli aggiuntivi che impediscono l'esecuzione del merge nel caso in cui l'evento rilevato non coinvolga direttamente le directory d'interesse.

Solo se la modifica riguarda effettivamente `batch-results/` o `batch-metrics/`, viene avviata la fase di verifica, in cui la funzione controlla che il numero complessivo di file presenti corrisponda al numero atteso, determinato sulla base della quantità di batch elaborati. Qualora tali condizioni risultino soddisfatte, viene attivato il processo di merge.

Quest'ultimo ha come obiettivo la generazione di:

- Un file finale contenente i risultati dell'analisi dell'intero dataset, salvato in formato JSON con nome `[dataset_name]_result.json` nella directory `results/`.
- Due file con le metriche aggregate, rispettivamente in formato JSON e CSV, salvati con i nomi `[dataset_name]_metrics.json` e `.csv` nella directory `metrics/`.

L'esistenza del merge handler è motivata dal fatto che il worker, ricevendo le richieste di elaborazione in parallelo per ciascun batch, non può scrivere direttamente su un unico file condiviso senza rischiare condizioni di race e corruzione dei dati.

Poiché il trigger generato da eventi GCS è asincrono e può attivare più istanze concorrenti del merge handler, è stato necessario implementare un meccanismo di lock distribuito. Questo avviene tramite la creazione di un file di controllo (`merge_lock.flag`) nella directory `control_flags/`. Solo la prima istanza che riesce a creare tale file procederà con l'aggregazione, mentre le altre interromperanno l'esecuzione. Il lock non viene rilasciato direttamente dal merge handler, ma dal server principale all'inizio di una nuova analisi, evitando così il rischio di esecuzioni ripetute o non intenzionali in seguito ad ulteriori eventi asincroni.

Affinché il merge venga eseguito correttamente, devono essere verificate due condizioni:

- la directory `batch-results/` non deve essere vuota, per escludere attivazioni causate da eventi su altre directory.
- il numero complessivo di file in `batch-results/` e `batch-metrics/` deve corrispondere al doppio del numero di batch previsti.

Solo al soddisfacimento di entrambe le condizioni viene tentata l'acquisizione del lock, seguita dall'unione dei file e dal caricamento dei risultati aggregati nel bucket.

3.3 Diagramma dell'architettura

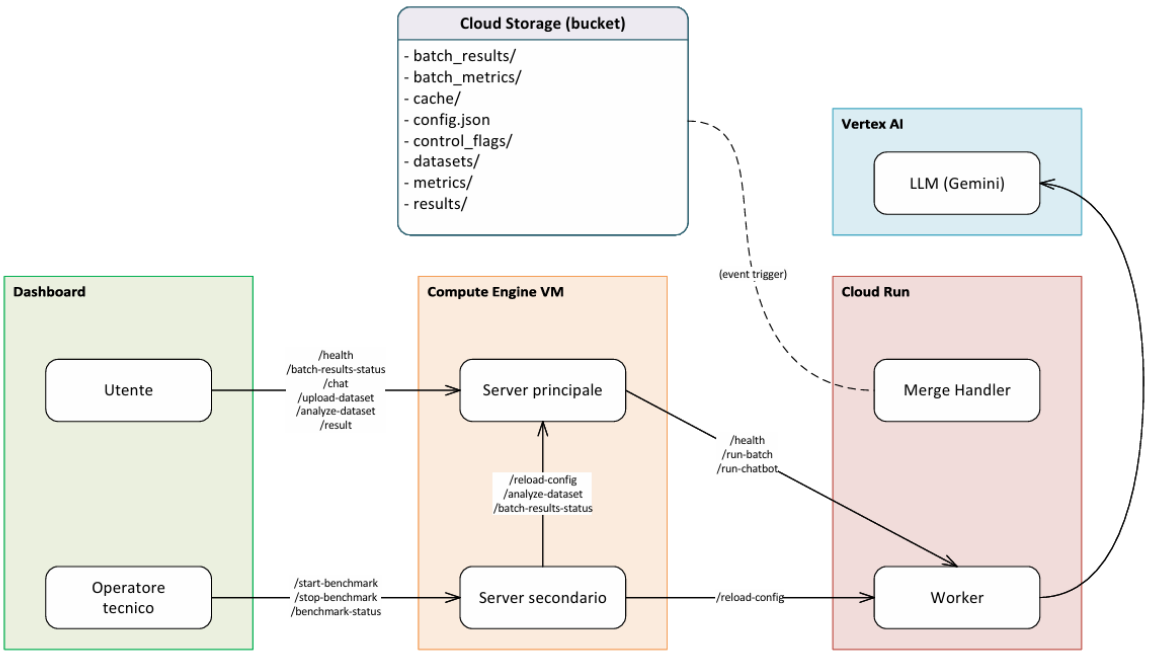

Figura 1: Architettura LLM4SOC

4 Analisi e classificazione degli alert

Il worker è il componente responsabile dell'analisi e della classificazione degli alert provenienti dal dataset precaricato sul bucket remoto in formato JSON, JSONL o CSV. Ciascun alert è descritto da un oggetto strutturato che contiene i seguenti campi: timestamp di generazione (time), etichetta identificativa (name), indirizzo IP (ip), hostname (host), codice di categoria (short), etichetta temporale time_label) ed etichetta dell'evento di sicurezza rilevato (event_label).

Per ciascun alert, il sistema costruisce dinamicamente un prompt ottimizzato per l'interazione con un LLM, nel caso specifico `gemini-2.0-flash-001`. Il prompt è progettato per richiedere al modello la classificazione dell'alert in una delle seguenti categorie: minaccia reale, falso positivo oppure indeterminato (nel caso in cui i dati non siano sufficienti per una valutazione). Viene inoltre richiesta una spiegazione testuale, leggibile e sintetica, che giustifichi la decisione fornita. A supporto del modello, è incluso nel prompt un esempio classificato, secondo un approccio few-shot.

```
ALERT:
{{
    "time": 1642213952,
    "name": "Wazuh: ClamAV database update",
    "ip": "172.17.131.81",
    "host": "mail",
    "short": "W-Sys-Cav",
    "time_label": "dirb",
    "event_label": "dirb"
}}

RISPOSTA:
{{
    "class": "false_positive",
    "explanation": "Aggiornamento del database ClamAV da host interno."
}}
```

L'elaborazione avviene in modalità concorrente per garantire scalabilità e reattività anche su dataset di grandi dimensioni. Il dataset viene suddiviso logicamente in batch di dimensioni fisse, dove ognuno viene gestito separatamente per ridurre l'impatto sulla memoria e migliorare la parallelizzazione. In fase di estrazione, viene evitato il caricamento completo del dataset in memoria: il file viene processato in streaming utilizzando il modulo `io.BytesIO`, da cui viene successivamente estratta solo la sezione corrispondente al batch d'interesse.

All'interno del batch, ciascun alert viene analizzato individualmente tramite una richiesta asincrona inviata al modello Gemini. Le chiamate concorrenti vengono orchestrate mediante il modulo `asyncio`, e il numero massimo di richieste simultanee è determinato da una variabile d'ambiente definita nel file di configurazione `config.json`. Il valore effettivamente utilizzato è calcolato come il minimo tra tale soglia e la dimensione del batch, evitando così di saturare inutilmente le risorse computazionali.

Le risposte restituite dal modello vengono convertite in oggetti JSON, ai quali vengono aggiunti metadati quali l'ID dell'alert e il relativo timestamp. In caso di errore nella chiamata o risposta non interpretabile, l'alert viene marcato con la classe `error` e viene associato un messaggio esplicativo.

Durante l'elaborazione di un batch, il worker calcola e registra un insieme di metriche di performance, salvate in remoto nella directory `batch-metrics/` del bucket GCS. Tra le metriche registrate vi sono:

- Il grado di parallelismo effettivamente utilizzato (parallelism_used);
- Il throughput espresso in alert al secondo (alert_throughput);
- Il consumo medio di memoria RAM (ram_mb);
- La durata totale di elaborazione del batch (time_sec);

- Il tempo medio di elaborazione di un alert (avg_time_per_alert);
- Il numero di classificazioni corrette (n_classified);
- Il tasso di successo e di errore (success_rate, error_rate);
- Il numero di errori complessivi (n_errors);
- La presenza o meno di errori (has_errors);
- Il numero di errori dovuti a timeout nelle richieste verso Gemini (n_timeouts).

Oltre a ciò, ciascun file di metrica contiene anche l'identificativo del batch (batch_id), il timestamp di avvio dell'elaborazione, la dimensione del batch e il valore di max_concurrent_reqs usato nella sessione.

Il worker include inoltre una funzionalità opzionale di caching che consente di memorizzare gli alert già classificati, al fine di evitare richieste ridondanti al modello. Tuttavia, nel contesto applicativo di questo progetto, l'utilizzo della cache non ha portato benefici significativi. La probabilità di incontrare lo stesso alert è infatti trascurabile e l'overhead introdotto dalla ricerca in cache, che cresce linearmente con la quantità di dati processati, risulta controproducente rispetto al guadagno teorico atteso. Per questo motivo, il meccanismo di caching è stato mantenuto a livello di codice, ma disabilitato in tutte le modalità operative del sistema, incluse le analisi standard e i benchmark.

5 Funzionamento della dashboard

La Dashboard rappresenta il fulcro dell'interazione utente all'interno del sistema *LLM4SOC*. Come anticipato nella sezione *3.1.1 Frontend React*, essa si occupa di gestire l'intero ciclo di vita dell'analisi, offrendo un'interfaccia reattiva e intuitiva che guida l'utente passo-passo nelle varie fasi operative: dal caricamento del dataset, alla visualizzazione e classificazione degli alert, fino all'esportazione e all'interrogazione del modello tramite chatbot.

Nella parte superiore della dashboard, l'utente trova il componente dedicato al caricamento del dataset. Attraverso un semplice pulsante di upload ("Scegli file"), viene consentito di selezionare un file locale in formato CSV, JSON o JSONL. Una volta caricato, viene avviato automaticamente il processo di analisi.

Figura 2: Avvio Dashboard

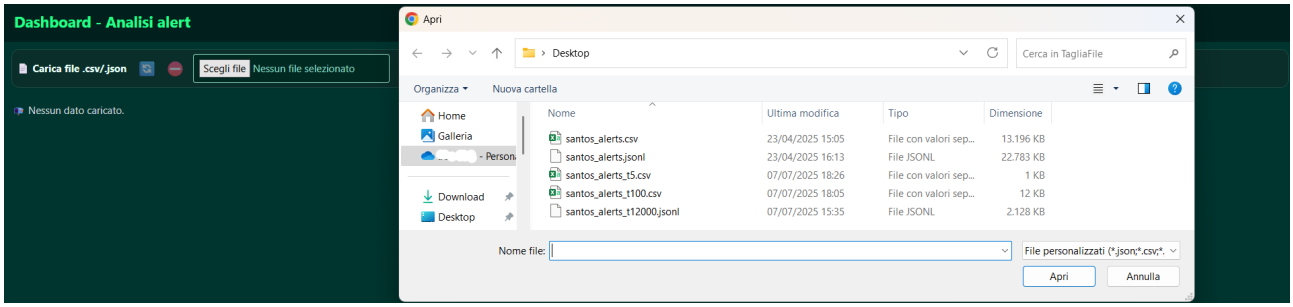

Figura 3: Selezione dataset

Durante l'elaborazione, la dashboard fornisce un feedback visivo costante: viene mostrato un timer con il tempo trascorso e il numero di tentativi effettuati per ottenere i risultati.

Figura 4: Attesa dei risultati

Terminata l'elaborazione, la tabella principale viene popolata dinamicamente con i risultati. Ogni riga rappresenta un alert, con colonne dedicate a: id, timestamp, class e explanation, ovvero la spiegazione generata dal modello LLM.

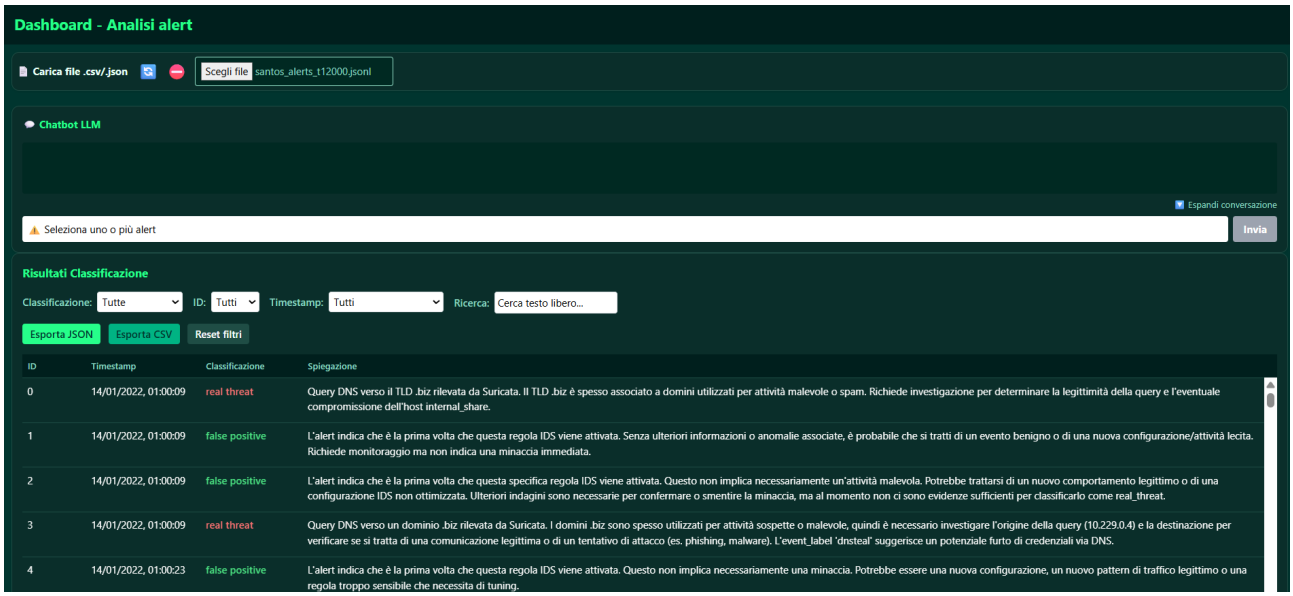

Figura 5: Alerts classificati

La tabella include strumenti di filtro avanzati che permettono di selezionare gli alert in base alla loro classificazione, il loro ID, al range temporale o tramite una barra di ricerca testuale per keyword.

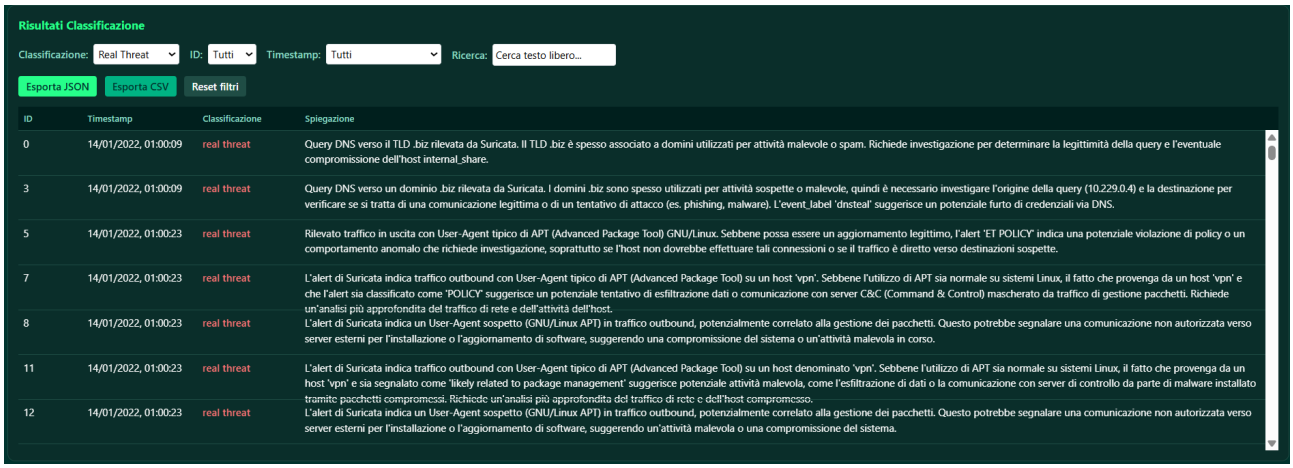

Figura 6: Filtro su "*real threat*"

Inoltre, la dashboard integra un chatbot interattivo, visibile solo dopo aver ricevuto i risultati e si attiva solamente dopo aver selezionato almeno un alert. Questo componente consente all'utente di inviare domande e ricevere risposte dal sistema LLM in linguaggio naturale. L'interfaccia "Chatbot LLM" visualizza in modo ordinato i messaggi scambiati, mantenendo lo storico fino a quando la sessione resta attiva.

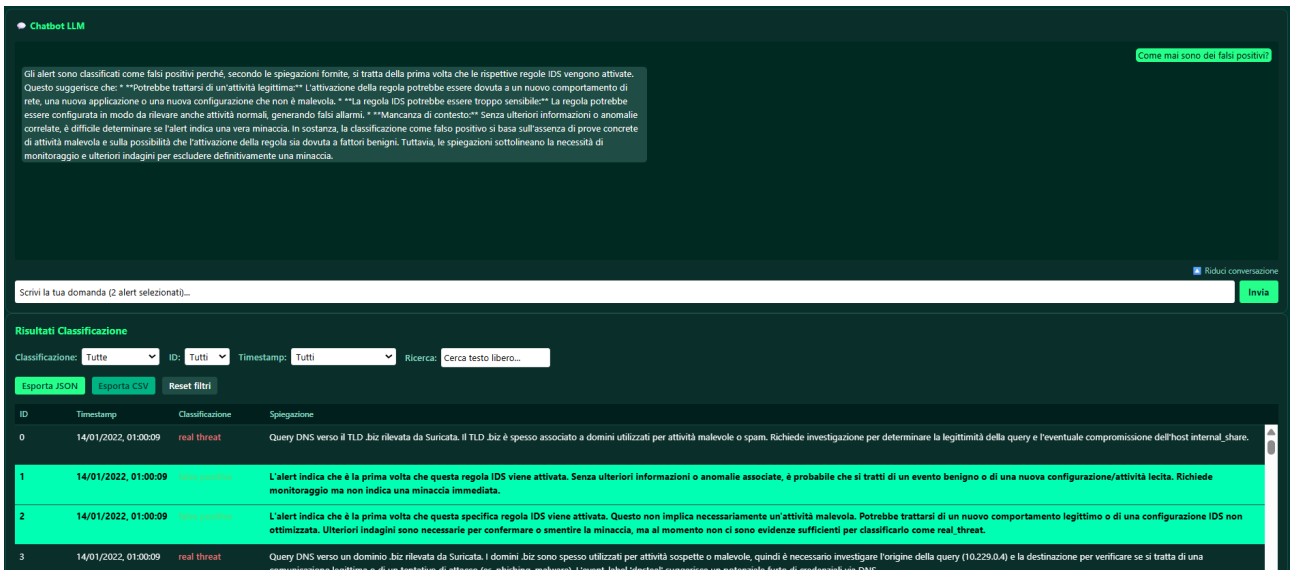

Figura 7: Chatbot

Infine, l'utente, dopo aver applicato determinati filtri, può esportare i dati in formato CSV o JSON. Questa funzionalità, accessibile direttamente dalla tabella, consente di salvare localmente sottoinsiemi specifici del dataset analizzato (es: tutti i falsi positivi rilevati, gli alert occorsi in un determinato intervallo temporale, ecc.) per successive analisi o attività di reportistica esterna.

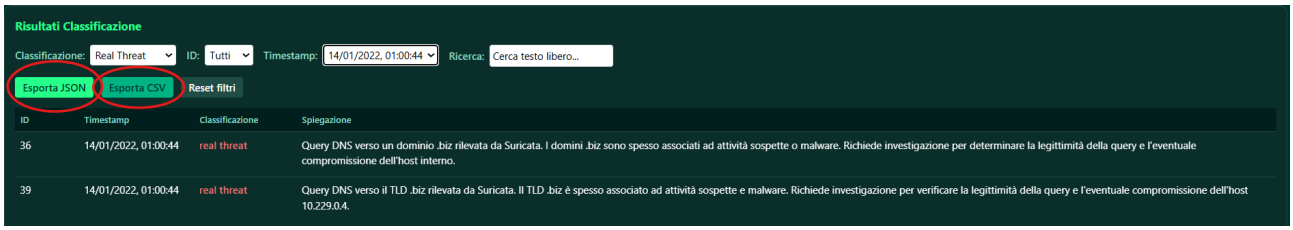

Figura 8: Export risultati

6 Benchmark e tuning parametrico

6.1 Benchmark

Il sistema include una funzionalità di benchmark automatizzato, inizialmente integrata all'interno dello stesso server FastAPI utilizzato per la gestione principale del backend. Tuttavia, l'esecuzione del benchmark comporta l'invio di richieste HTTP al server stesso, generando una condizione di deadlock se le chiamate partono e si attendono risposta dallo stesso processo. Per risolvere questa criticità, la logica di benchmark è stata estratta e isolata in un secondo server FastAPI, eseguito come processo separato sulla medesima macchina virtuale, ma in ascolto su una porta diversa. Questa separazione logica consente ai due server di operare in parallelo, evitando interferenze e mantenendo l'architettura modulare e stabile.

Una volta avviato, il processo esegue ripetutamente l'analisi di un dataset specificato in input, esplorando in modo sistematico tutte le combinazioni possibili dei parametri batch_size e max_concurrent_reqs all'interno dei rispettivi intervalli di configurazione. L'obiettivo del benchmark è raccogliere un insieme sufficientemente ampio di metriche, da cui generare un dataset utile all'addestramento di un modello di machine learning in grado di suggerire, in modo automatico, la configurazione ottimale per l'elaborazione degli alert, bilanciando prestazioni e utilizzo delle risorse computazionali.

L'avvio del benchmark avviene tramite una richiesta all'endpoint `/start-benchmark`, alla quale è possibile fornire, oltre al nome del dataset, parametri opzionali che definiscono gli intervalli e i passi di incremento per i valori di batch_size e max_concurrent_reqs. In risposta, il sistema restituisce un messaggio di conferma dell'avvio del processo, che viene eseguito in background.

L'esecuzione può essere interrotta anticipatamente tramite l'endpoint `/stop-benchmark`, che genera un file di controllo denominato `benchmark_stop.flag` all'interno di una directory dedicata. Il processo di benchmark verifica ciclicamente la presenza di tale file e, qualora lo rilevi, lo interpreta come segnale di terminazione, interrompendo in modo ordinato l'elaborazione in corso.

Lo stato di avanzamento può essere monitorato in tempo reale tramite l'endpoint `/benchmark_status`, che restituisce un oggetto JSON aggiornato dinamicamente, contenente i parametri correnti e i metadati associati al benchmark. In particolare, il JSON include i seguenti campi:

- Dimensione del batch corrente (batch_size);
- Numero massimo di richieste concorrenti inviate al modello Gemini (max_concurrent_requests);
- Stato attuale del benchmark (status);
- Timestamp dell'ultima modifica al file di stato (last_updated);
- Nome completo del dataset in uso (dataset_filename);
- Passo di incremento per ciascuna delle due variabili (batch_size_step e max_reqs_step);
- URL dell'ultimo endpoint invocato (last_request);
- Numero totale di alert presenti nel dataset (tot_alerts).

Il benchmark si articola in tre fasi principali. Durante l'inizializzazione vengono rimossi eventuali flag generati da esecuzioni precedenti, viene creato un backup del file di configurazione ambientale e aggiornato il file di contesto. La fase di esecuzione consiste in due cicli annidati che incrementano progressivamente batch_size e max_concurrent_reqs, avviando ad ogni iterazione una nuova richiesta di analisi e monitorandone l'avanzamento tramite l'endpoint `/batch-results-status`. La fase finale ripristina il file di configurazione originale e registra le ultime informazioni sul contesto.

6.2 Tuning

I dati raccolti al termine del benchmark, salvati in formato CSV, sono stati utilizzati per addestrare un modello di regressione volto a stimare il tempo di esecuzione in funzione dei parametri forniti. La variabile target scelta è *time_sec*, corrispondente alla durata, in secondi, dell'elaborazione di ciascun batch da parte del worker. Le feature utilizzate per la predizione includono, tra le altre, batch_size e max_concurrent_reqs, che rappresentano le variabili di interesse da ottimizzare.

La fase preliminare di analisi esplorativa è stata condotta mediante l'utilizzo combinato di boxplot, pairplot e matrici di correlazione, al fine di studiare la distribuzione delle variabili e le relazioni tra le diverse feature (vedi anche Figura 13, 14 e 15). È emersa fin da subito l'assenza di contenuto informativo nelle metriche relative agli errori: durante l'intero processo di benchmark non si sono infatti verificati fallimenti, rendendo tali variabili prive di contenuto informativo e quindi inutili ai fini del tuning. Di conseguenza, queste feature sono state rimosse, insieme ad altre ridondanti come batch_id e timestamp, la cui informazione non contribuisce alla qualità del modello. Inoltre, sono state escluse dal dataset finale anche le coppie di variabili caratterizzate da una forte correlazione lineare, in quanto da due feature fortemente correlate è possibile derivare le stesse informazioni.

Una particolare osservazione emersa dall'analisi del pairplot (Figura 9) riguarda la relazione tra la variabile target time_sec e la feature batch_size, la quale risulta biforcata. Tale biforcazione è dovuta alla presenza di due comportamenti distinti all'interno del dataset: il ramo superiore del grafico corrisponde alle analisi iniziali eseguite con max_concurrent_reqs pari a 1, ovvero in modalità sequenziale. In questi casi, le richieste verso il LLM vengono inviate una alla volta, vanificando i benefici dell'architettura asincrona basata su `asyncio`.

Questi dati, rappresentativi di una configurazione poco realistica e non ottimizzata, introducono rumore nel modello predittivo e sono stati quindi esclusi dal dataset finale per preservarne la coerenza e migliorare l'accuratezza del modello predittivo.

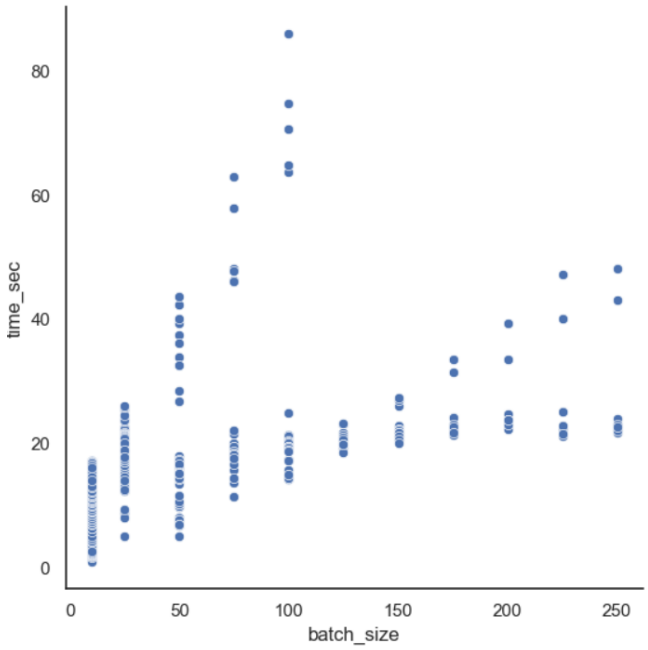

Figura 9: Pairplot - Relazione tra *time_sec* e *batch_size*

Una volta completata la fase di pulizia del dataset, è stata eseguita una regressione polinomiale con l'obiettivo di modellare l'andamento della variabile *time_sec* in funzione di batch_size e max_concurrent_reqs. Questo ha permesso di identificare con maggiore chiarezza le regioni di configurazioni parametriche più efficienti in termini di tempo d'esecuzione.

I grafici generati a partire dal modello di regressione (Figura 10) descrivono l'andamento della funzione obiettivo nello spazio dei parametri, consentendo di individuare le configurazioni che offrono migliori prestazioni in termini di tempo di esecuzione.

Nel primo grafico, che mostra la relazione tra il tempo di esecuzione (time_sec) e la dimensione del batch (batch_size), si osserva un fitto cluster di punti in prossimità dell'origine: questo comportamento è dovuto alla scelta di valori molto bassi per batch_size, che causano un'elevata frammentazione del carico di lavoro. In tali condizioni, il sistema genera un numero elevato di task concorrenti, saturando le code gestite dal servizio Cloud Tasks e introducendo una variabilità significativa nei tempi di risposta. Poiché solo una parte delle richieste può essere gestita immediatamente, mentre le restanti vengono inserite in coda, si verifica un disallineamento nei tempi di risposta: le richieste accodate accumulano ritardo, contribuendo a un incremento significativo della latenza media. Alla luce di questa osservazione, risulta consigliabile adottare un valore di batch_size pari o superiore a 50. All'estremo opposto del grafico, a partire da valori di batch_size prossimi a 150, si manifesta un andamento crescente del tempo di esecuzione, più accentuato nei casi in cui il parallelismo è limitato. Questo fenomeno è attribuibile ai colli di bottiglia computazionali generati dalla gestione di batch di grandi dimensioni in assenza di un numero adeguato di richieste concorrenti.

Il secondo grafico analizza la relazione tra time_sec e max_concurrent_reqs. È evidente che valori molto bassi, in particolare max_concurrent_reqs pari a 1, annullino completamente i benefici offerti dall'architettura asincrona del worker basata sul modulo `asyncio`, forzando l'elaborazione sequenziale degli alert. Per valori più elevati, la densità dei punti e la sovrapposizione delle osservazioni non consentono di trarre conclusioni definitive dal grafico bidimensionale. Per questo motivo, è stato utile ricorrere a una rappresentazione tridimensionale, in grado di sintetizzare efficacemente le interazioni tra i due parametri di input e il tempo di esecuzione.

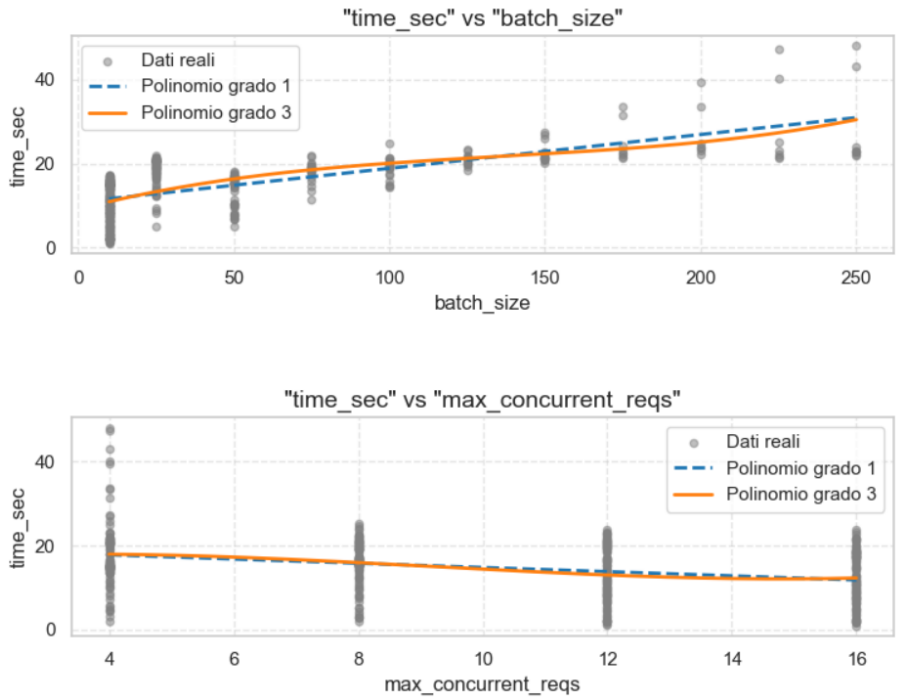

Figura 10: Regressioni polinomiali su `time_sec` rispetto a *batch_size* e *max_concurrent_reqs*

L'analisi tridimensionale della superficie di regressione (Figura 11) ha permesso di osservare la variazione congiunta della funzione obiettivo rispetto ai due parametri batch_size e max_concurrent_reqs. Le aree scure, corrispondenti ai tempi di esecuzione più contenuti, si concentrano visibilmente attorno a valori intermedi di batch_size e a valori elevati di max_concurrent_reqs. A questo punto è possibile cominciare a delineare dei range ottimali per la scelta dei parametri: alla luce delle osservazioni emerse anche nei grafici bidimensionali precedenti, il sistema dimostra di gestire in modo relativamente efficiente batch contenenti tra i 50 e i 150 alert, associati a un parallelismo compreso tra 8 e 16 richieste simultanee inviate al LLM.

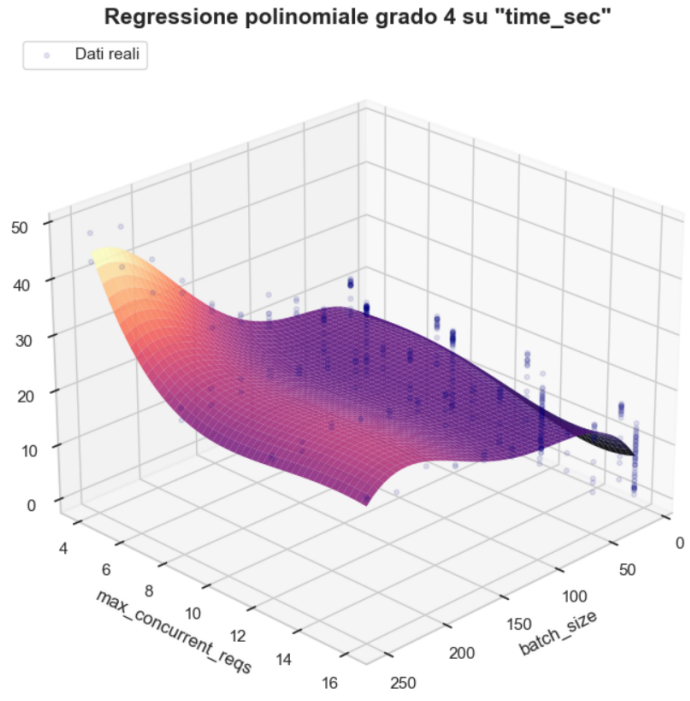

Figura 11: Regressione polinomiale di grado 4 su *time_sec* in funzione di *batch_size* e *max_concurrent_reqs*

Infine, per non determinare gli intervalli ottimali unicamente sulla base di un'osservazione visiva, si è rafforzata l'analisi in modo più oggettivo, tramite una pivot table (Figura 12) che riassume le medie dei tempi di elaborazione associati a ciascuna combinazione dei parametri considerati.

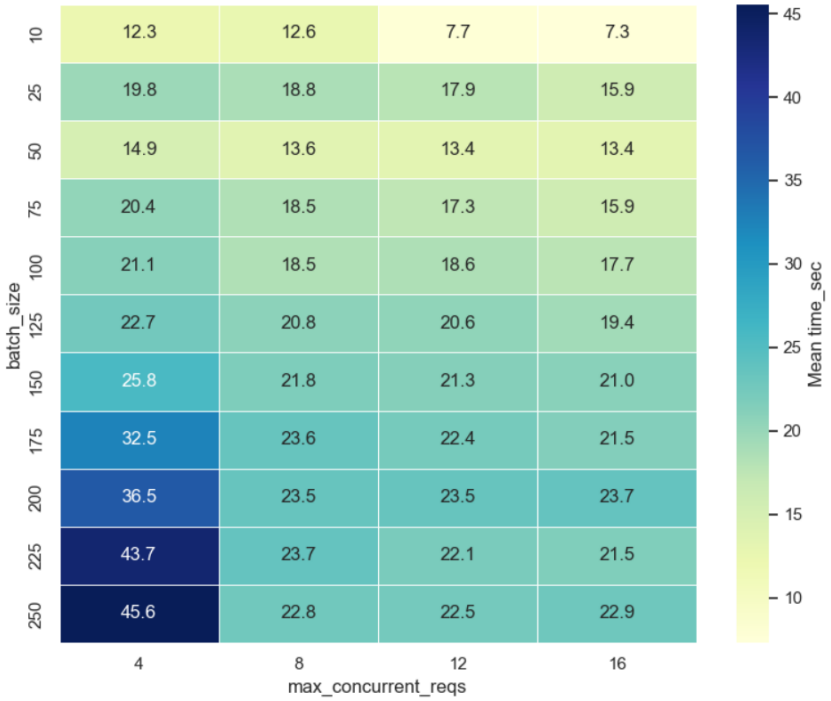

Figura 12: Pivot Table - Tempi medi d'elaborazione (*time_sec*) per combinazioni di parametri indipendenti

I risultati confermano che le configurazioni più performanti si collocano nell'intersezione tra valori intermedi di batch_size e valori elevati di max_concurrent_reqs.

Si può quindi concludere che, sulla base dell'analisi condotta, i range che garantiscono un buon compromesso tra parallelismo e carico computazionale risultano essere:

- batch_size $\in$ [50, 100]
- max_concurrent_reqs $\in$ [8, 16]

7 Conclusioni

Il progetto *LLM4SOC – False Positive Triage Assistant for IDS Alert Analysis* ha dimostrato la fattibilità di un sistema automatizzato in grado di supportare il processo di analisi e classificazione degli alert di sicurezza generati da un IDS. Grazie all'impiego di un modello linguistico avanzato (LLM), integrato in un'infrastruttura cloud scalabile su Google Cloud Platform e gestita tramite Terraform, è stato possibile costruire un prototipo funzionante che unisce automazione, spiegabilità e semplicità d'uso attraverso una dashboard interattiva.

Durante la fase di valutazione delle prestazioni (Capitolo 6 - Benchmark e tuning parametrico), la componente di benchmark parametrico ha permesso di identificare range ottimali per i parametri batch_size e max_concurrent_reqs, rispettivamente compresi tra 50 e 100 per il primo, e tra 8 e 16 per il secondo. Queste configurazioni hanno evidenziato un buon compromesso tra parallelismo e carico computazionale, fornendo un primo livello di tuning automatico in grado di bilanciare throughput e latenza.

Lavori futuri Durante lo sviluppo del progetto sono emerse alcune possibili evoluzioni per migliorare l'efficienza e l'estensibilità del sistema.

Un primo spunto riguarda il meccanismo di caching, attualmente disabilitato nel worker, che potrebbe essere ripensato tramite tecniche più avanzate, come l'hashing degli alert o il riconoscimento automatico di pattern ricorrenti, al fine di ottimizzare la gestione degli alert duplicati.

Un ulteriore miglioramento consiste nell'ampliare il supporto ai formati di input, includendo ad esempio XML, Parquet o stream di dati, per aumentare la compatibilità del sistema con ambienti SIEM e pipeline più complesse.

Anche la componente di interazione con il LLM potrebbe essere potenziata attraverso l'uso di prompt dinamici o mediante fine-tuning su dati reali, con l'obiettivo di migliorare la precisione delle classificazioni e la qualità delle spiegazioni.

Infine, un'estensione interessante potrebbe essere l'integrazione di un modulo di analisi predittiva, in grado di anticipare la classificazione degli alert prima della validazione manuale. In parallelo, si potrebbe prevedere la visualizzazione sulla dashboard di una stima dei tempi di elaborazione, utile per monitorare le performance in fase di analisi.

Appendice

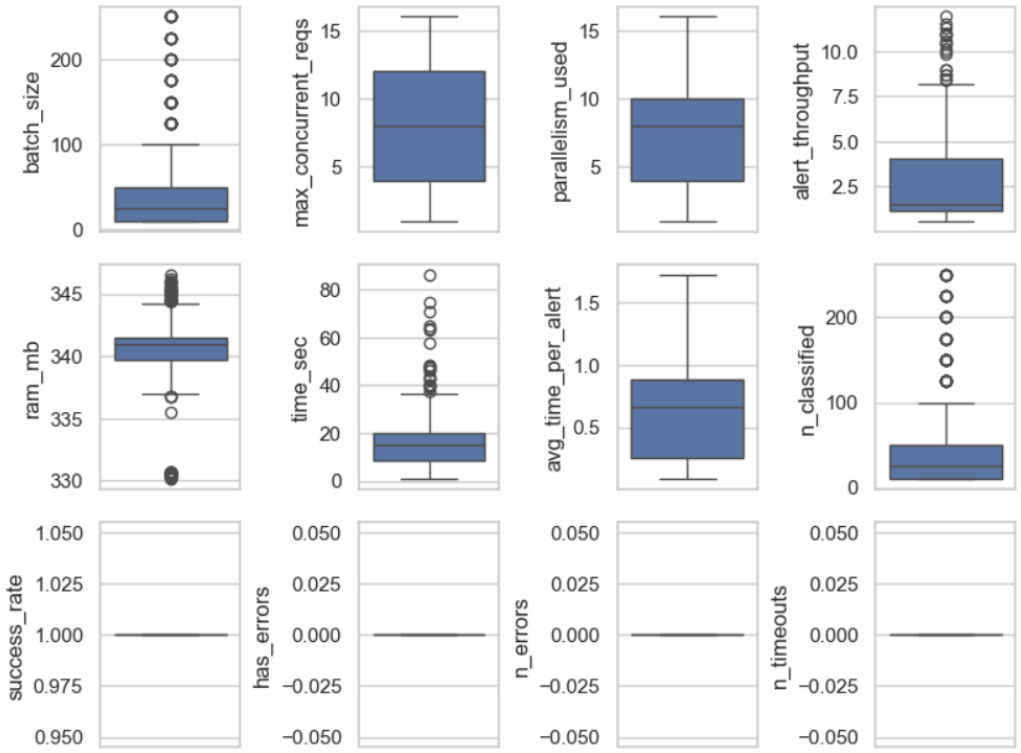

Figura 13: Boxplot - Distribuzioni delle feature

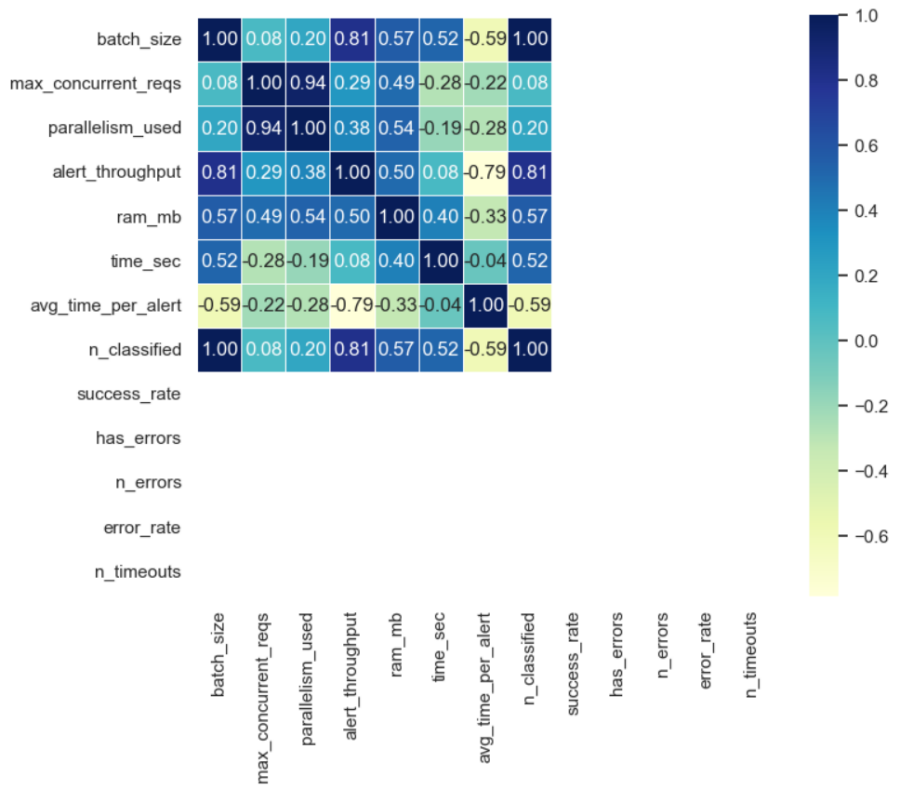

Figura 14: Matrice di correlazione

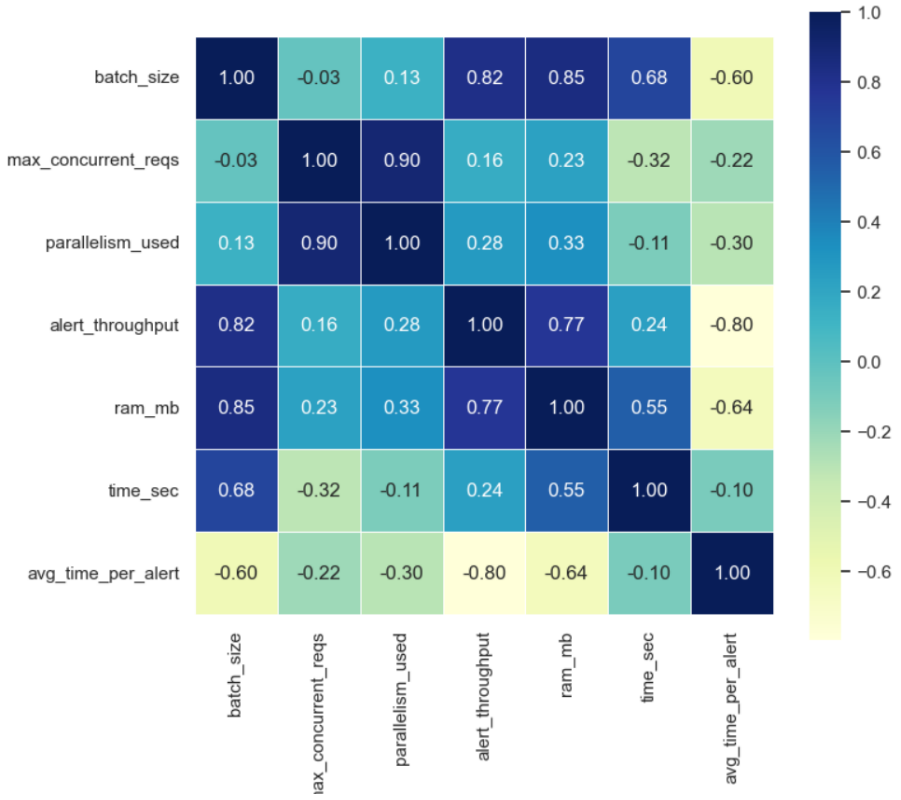

Figura 15: Matrice di correlazione (ripulita da feature costanti)