Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Progettazione di un Adapter software per il calcolo della Carbon Footprint di applicazioni cloud

Relatore:
Antonio Brogi

Tutore aziendale:
Dario Frongillo

Candidato:
Flavio Romano

Anno Accademico 2023/2024

Indice

Capitolo 1

Introduzione

1.1 ICT e sostenibilità

L'inquinamento ambientale rappresenta una delle sfide più urgenti e complesse che l'umanità deve affrontare nel XXI secolo. L'uomo sta avendo un impatto sempre più gravoso sull'ambiente naturale, mettendo a rischio gli ecosistemi e la stessa sopravvivenza di numerose specie, compresa la propria. Secondo il report del 2023 delle Nazioni Unite sui Cambiamenti Climatici, le emissioni globali di gas serra hanno raggiunto livelli record, contribuendo ad un aumento della temperatura media globale di circa 1,1°C rispetto all'era preindustriale [1]. Gli effetti del riscaldamento globale sono ormai evidenti ovunque: dallo scioglimento dei ghiacciai all'innalzamento del livello dei mari, dalle sempre più frequenti ondate di calore alle condizioni meteorologiche estreme. Ma l'inquinamento non riguarda solo le emissioni di CO2. La deforestazione, l'eccessivo sfruttamento delle risorse naturali e l'accumulo di rifiuti plastici negli oceani stanno causando una perdita drammatica di biodiversità. L'Accordo di Parigi del 2015 ha fissato l'obiettivo di limitare il riscaldamento globale sotto la soglia di 1,5°C, impegnando i paesi firmatari a ridurre progressivamente le emissioni di gas serra [2].

L'anidride carbonica

L'anidride carbonica, comunemente nota come CO2, è un gas naturalmente presente nell'atmosfera terrestre. Viene prodotta da una vasta gamma di processi naturali come l'erosione delle rocce, le eruzioni vulcaniche e la respirazione degli organismi viventi. Tuttavia, le attività umane legate alla combustione di combustibili fossili come carbone, petrolio e gas naturale hanno fatto aumentare in modo allarmante le concentrazioni di CO2 nell'atmosfera rispetto ai livelli pre-industriali. Negli ultimi decenni l'eccesso di anidride carbonica nell'atmosfera rappresenta uno dei principali fattori alla base del riscaldamento globale e dei cambiamenti climatici. La CO2 è infatti un gas serra, ovvero assorbe e trattiene il calore proveniente dalla radiazione infrarossa emessa dalla superficie terrestre, impedendone la dispersione nello spazio e quindi viene intrappolato più calore nell'atmosfera [3].

Tra le principali fonti di inquinamento ci sono:

L'industria energetica dipendente dai combustibili fossili, con un impatto devastante sull'ambiente marino e sulla salute umana.

L'agricoltura intensiva con la deforestazione e le emissioni di metano, contribuisce in modo significativo alle emissioni globali di gas serra.

L'industria tessile alimentata dalla fast fashion, è responsabile di enormi quantità di rifiuti e di emissioni di carbonio.

Il settore dei trasporti è uno dei maggiori responsabili dell'inquinamento a causa della sua forte dipendenza dai combustibili fossili. L'utilizzo di automobili, camion, navi e aerei alimentati da benzina, diesel e cherosene rilascia nell'atmosfera enormi quantità di gas serra.

C'è un ultimo settore che spesso passa in sordina quando si parla di inquinamento, **l'industria IT** (*Information Technology*). Mentre le nostre vite ruotano sempre più attorno alla tecnologia digitale, le preoccupazioni sull'impatto ambientale della digitalizzazione sono cresciute. I progressi nel settore tecnologico continuano a rivoluzionare il modo in cui viviamo, lavoriamo e comunichiamo, ma hanno un costo. Dall'enorme quantità di energia necessaria per alimentare i data center allo smaltimento dei rifiuti elettronici. Secondo l'ONU, l'industria tecnologica attualmente rappresenta il 2-3% delle emissioni globali e si prevede che aumenti rapidamente man mano che la digitalizzazione prende piede. Il contributo dell'industria tecnologica alle emissioni di gas serra continuerà ad aumentare se non affrontato, con i software utilizzati dalle aziende responsabili dell'emissione di 350-400 megatoni[1] di anidride carbonica da soli [5].

Con l'onnipresenza della tecnologia nella nostra vita quotidiana, la crescita esponenziale della potenza di calcolo nei data center, nel cloud, nei dispositivi elettronici e nei servizi digitali ha determinato un aumento significativo della domanda di energia e delle emissioni di gas serra associate alle operazioni IT. L'industria tecnologica, infatti, consuma enormi quantità di energia per alimentare le sue infrastrutture, con i data center che da soli rappresentano il 45% delle emissioni di gas serra nel settore ICT globale. La forte dipendenza da fonti energetiche prevalentemente fossili contribuisce in modo significativo all'impronta di carbonio del settore e, di conseguenza, al cambiamento climatico.

Classificazione delle emissioni

Il GHG (Greenhouse Gas) Protocol è uno tra gli stardard più noti a livello mondiale per misurare e gestire le emissioni di CO2. Ha creato standard contabili, strumenti e formazione per aiutare le imprese a misurare e gestire le emissioni climalteranti. Il GHG Protocol Corporate Standard classifica le emissioni di gas a effetto serra associate alla Corporate Carbon Footprint (CCF) di un'azienda come emissioni di *scope* (ambiti) diversi, suddivide le emissioni in tre categorie principali, ognuna delle quali rappresenta un diverso livello di controllo e responsabilità da parte dell'azienda:

[1] Un megatone di CO2 equivale a un miliardo di chilogrammi di CO2.

Scope 1: Emissioni dirette generate direttamente dalle attività dell'azienda e dalle risorse che possiede o controlla. e.g emissioni dei veicoli aziendali, emissioni di gas refrigeranti.

Scope 2: Emissioni indirette generate indirettamente dall'azienda, ma derivano dal consumo di energia acquistata da fornitori esterni, come l'elettricità utilizzata per alimentare uffici o stabilimenti. Sebbene non siano prodotte direttamente dall'azienda, sono comunque una conseguenza delle sue attività e rientrano nella sua responsabilità.

Scope 3: Altre emissioni indirette questa è la categoria più ampia e comprende tutte le altre emissioni indirette lungo l'intera catena del valore dell'azienda. L'agenzia statunitense EPA descrive questa categoria di emissioni come "il risultato di attività provenienti da beni non posseduti o controllati dall'organizzazione che redige il bilancio, ma che l'organizzazione impatta indirettamente nella sua catena del valore". Anche se queste emissioni sono fuori dal controllo dell'azienda che redige il bilancio, possono rappresentare la parte più consistente del suo inventario di emissioni di gas serra.

Il GHG Protocol suddivide le emissioni Scope 3 in due macro-categorie, a monte (*upstream*) e a valle (*downstream*), in base al loro posizionamento nella catena del valore dell'azienda.

Emissioni a monte comprendono tutte le emissioni indirette generate prima che i prodotti o servizi entrino nell'organizzazione. Si tratta di emissioni legate all'acquisizione di beni e servizi, dalla loro produzione fino al momento in cui vengono consegnati all'azienda. e.g emissioni legate alla pruduzione e trasporto di macchinari acquistati dall'azienda, emissioni legate allo smaltimento dei rifiuti prodotti dai fornitori dell'azienda, emissioni generate dal tragitto casa-lavoro dei dipendenti dei fornitori dell'azienda.

Emissioni a valle comprendono tutte le emissioni indirette generate dopo che i prodotti o servizi hanno lasciato l'organizzazione. Si tratta di emissioni legate all'utilizzo, al trasporto, alla trasformazione e allo smaltimento dei prodotti venduti dall'azienda. e.g emissioni derivanti da ulteriori trasformazioni o lavorazioni dei prodotti venduti dall'azienda, effettuate da altre organizzazioni.

La promessa delle aziende

Il termine **Net Zero** si riferisce all'equilibrio tra la quantità di gas a effetto serra (GHG) rilasciati nell'atmosfera e la quantità di gas a effetto serra rimossi, una neutralità carbonica. Le Nazioni Unite definiscono Net Zero

> La riduzione delle emissioni di gas a effetto serra il più vicino possibile allo zero, con il riassorbimento delle emissioni rimanenti dall'atmosfera, dagli oceani e dalle foreste

L'obiettivo di limitare al di sotto di 1,5°C il riscaldamento globale e di raggiungere emissioni nette di carbonio pari a zero è stato previsto dall'Accordo di Parigi, un

trattato giuridicamente vincolante firmato da 196 Paesi [2]. Anche molte aziende IT hanno preso un impegno simile, fra queste Apple è una delle più impegnate e trasparenti, promette di raggiungere Net Zero per tutta la sua filiera e i suoi prodotti entro il 2030.

Nel report del 2022 Environmental Progress Report [6] la Apple descrive la sua impronta carbonica del 2021 attraverso questo grafico:

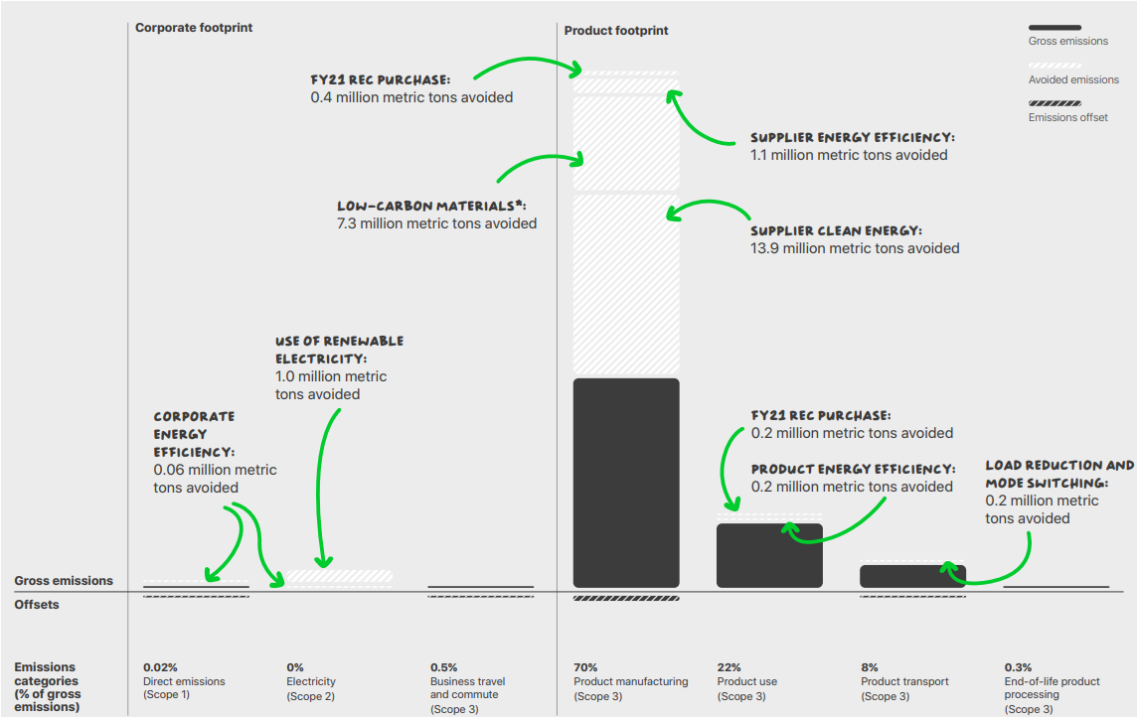


Figura 1.1: Grafico sulla carbon footprint del 2021 di Apple, le emissioni nette di carbonio di Apple nel 2021 erano di 22,5 milioni tonnellate di CO2

dal grafico si evince che ha raggiunto la neutralità carbonica per le emissioni aziendali: per emissioni dirette (scope 1) e emissioni indirette legate all'elettricità (scope 2). Per quanto riguarda le altre emissioni indirette (scope 3), Apple ha azzerato le sole emissioni derivanti dai viaggi di lavoro e dal tragitto casa-lavoro dei dipendenti, ma non basta. Net0Tracker la colloca alla nona posizione mondiale per impegno nel raggiungere la Net Zero, tuttavia ancora la strada è molto lunga [7]. Perché ridurre le emissioni in scope 3 significherebbe:

- Ridurre le emissioni legate alla produzione dei dispositivi elettronici (computer, smartphone, server) e al loro trasporto lungo la catena di approvvigionamento.

- Ridurre le emissioni generate dai prodotti durante il loro ciclo di vita

per una IT company come Apple è una sfida veramente difficile, visto che significherebbe effettuare cambiamenti radicali all'interno della catena di produzione.

1.2 NTT DATA e Green Software Foundation

NTT DATA è una multinazionale che si occupa di *system integration* e consulenza facente parte del gruppo giapponese NTT (*Nippon Telegraph and Telephone*). Nel 2021 NTT DATA entra a far parte - come membro del comitato direttivo - della Green Software Foundation (GSF), un'organizzazione dedicata alla riduzione delle emissioni di CO2 causate dal software. La Green Software Foundation si è posta come obiettivo una riduzione del 45% delle emissioni di CO2 nel settore IT entro il 2030, mentre in questo ambito NTT DATA sta lavorando a degli standard di sviluppo, pratiche per ridurre le emissioni di CO2 e strumenti per il calcolo dei volumi di emissioni di CO2 prodotti dal software.

Software Carbon Intensity (SCI)

La Green Software Foundation ha creato la metrica Software Carbon Intensity (SCI), è una misura della quantità di emissioni di carbonio associate alla creazione, all'esecuzione e all'utilizzo di software. Tiene conto di diversi fattori:

Consumo energetico del server La quantità di energia consumata dai server che eseguono il software. Questo dipende dal carico, dalla potenza di elaborazione richiesta e dalla durata dell'esecuzione del software.

Efficienza energetica del data center : I data center che ospitano i server possono variare notevolmente in termini di efficienza energetica. Alcuni utilizzano fonti di energia rinnovabile o tecnologie di raffreddamento più efficienti, riducendo l'impatto ambientale complessivo.

Efficienza del codice Codice più efficiente richiede meno risorse di elaborazione e quindi genera minori emissioni di carbonio.

Inoltre la GSF ha definito una specifica (SCI Specification) per calcolare il tasso di emissioni di carbonio di un sistema software, attraverso cui misura l'impronta di carbonio di un software in termini di emissioni di CO2 *equivalenti*. L'equazione di base per calcolare il tasso di emissioni di carbonio di un sistema software è la seguente:

$$\begin{aligned} \text{SCI} &= C \text{ per } R \\ &= (O + M) \text{ per } R \\ &= [(E \cdot I) + M] \text{ per } R \end{aligned}$$

C: Emissioni di carbonio

Sono le emissioni di carbonio misurate in gCO2e, può essere ulteriormente descritto come la somma di due tipi principali di emissioni: $O + M$

O: emissioni operative sono emissioni causate dal consumo energetico del software in uso. Possono variare notevolmente a seconda dell'efficienza del software, dell'hardware su cui viene eseguito e delle fonti energetiche utilizzate per alimentare l'infrastruttura. Per calcolare le emissioni operative, è necessario

considerare il consumo energetico del software durante il suo funzionamento e moltiplicarlo per la carbon intensity dell'elettricità utilizzata per alimentare l'hardware, otteniamo quindi: $E \cdot I$

E: energia consumata è l'energia consumata dal software durante il suo funzionamento, misurata in kilowattora kWh.

I: carbon intensity è una misura di quanto sia pulita l'elettricità utilizzata. Si riferisce a quanti grammi di anidride carbonica (CO2) vengono rilasciati per produrre un chilowattora (kWh) di elettricità, infatti l'unità di misura è gCO2e/kWh.

M: embodied emissions sono le emissioni di gas serra generate durante l'intero ciclo di vita di un prodotto o di un servizio. Quindi le emissioni generate durante i processi di produzione, trasporto, utilizzo e smaltimento del prodotto.

La CO2e (anidride carbonica equivalente) viene utilizzata come unità di misura per semplificare la misurazione e il confronto dell'impatto di diversi gas serra (considerati nel valore di CO2 ad essi equivalenti) sul riscaldamento globale.

R: Functional Unit

E' un'unità di riferimento, rappresenta come scala il software e viene scelta in base al tipo di applicazione. e.g per un API potrebbe essere l'esecuzione di una singola chiamata o il trasferimento di un certo volume di dati in un ora, per un sito web potrebbe essere il numero di login giornalieri. La functional unit permette di

- normalizzare le misurazioni, rendendo confrontabili emissioni di CO2 di diversi software indipendentemente dalla loro complessità.
- identificare le aree di miglioramento, individuando quali fasi del ciclo di vita o funzionalità del software contribuiscono maggiormente alle emissioni.

quindi all'aumento del carico di lavoro dell'applicazione corrisponderà un'aumento del valore di R.

Green IT

Green IT è una suite *plug and play* di strumenti utili per calcolare l'impronta di carbonio generata dai software partendo dall'energia consumata. I dati vengono raccolti e inviati al cliente, oppure possono essere visualizzati su una dashboard accendendo al proprio profilo. I due principali KPI sono la CO2 e lo SCI. La CO2 è proprio la C vista prima

$$C = E \cdot I$$

dove E è l'energia consumata in kWh e I è la carbon intensity misurata in gCO2e/kWh.

Gli strumenti che mette a disposizione NTT DATA permettono di stimare l'energia consumata utilizzando le metriche dell'hardware su cui gira il software. In particolare sono state utilizzate le metriche per CPU, GPU, memoria e storage.

1.3 Obiettivo del tirocinio

L'obiettivo di questo tirocinio è quello di progettare e implementare l'adapter (o connettore) per Google Cloud. Consideriamo un cliente che ha un sistema deployato su GCP, una volta che ha definito un JSON di configurazione e caricato quest'ultimo su Google Cloud Storage, il connettore:

1. raccoglierà le metriche di utilizzo dall'API di monitoraggio di GCP
2. trasformerà i dati
3. li invierà a Green IT per stimare le emissioni di CO2

I requisiti principali del connettore sono:

- poter leggere un JSON di configurazione caricato su Google Cloud Storage
- raccogliere le metriche per i layer: Cloud Run, Cloud SQL, Compute Engine, GKE
- formattare le metriche contenute secondo le specifiche dell'API di Green IT
- poter definire in maniera semplice nuove Functional Unit
- essere altamente configurabile attraverso variabili d'ambiente
- supportare due modalità di avvio: una con invio di metrica e l'altra senza
- inviare le metriche a Green IT

1.4 Risultati ottenuti

Lo sviluppo del connettore è stato completato con successo in anticipo rispetto alla fine del tirocinio, il tempo rimasto è stato usato per documentare i comportamenti di Green IT in situazioni di carico, a cui ha risposto molto bene. Il connettore è servito come base per lo sviluppo di nuovi connettori software in grado di interagire con diversi Cloud Provider (ad esempio, AWS). Grazie al lavoro sul connettore, è stato possibile certificare Green IT come software green. Il connettore che ho creato è in grado di:

- ottenere le metriche dei componenti Google Cloud Platform di Green IT
- inviare le metriche a due endpoint diversi
- calcolare il valore di Functional Unit per lo SCI

Inoltre, ho redatto la documentazione completa del progetto, garantendo la continuità dello sviluppo e facilitando il passaggio di consegna ad altri sviluppatori.

Capitolo 2

Tecnologie utilizzate

2.1 Google Cloud Platform

Google Cloud Platform (GCP) è una delle principali piattaforme di cloud computing a livello globale e offre un'ampia gamma di servizi e funzionalità. GCP libera l'azienda dalla gestione dell'infrastruttura, dal server e dalla configurazione reti. Essa offre Infrastructure as a service, Platform as a service ed ambienti serverless.

IaaS

IaaS (Infrastructure as a Service) è un modello di servizi cloud che offre risorse di infrastruttura on demand, come calcolo, archiviazione, networking e virtualizzazione, ad aziende e privati tramite il cloud. IaaS è molto conveniente perché l'acquisizione di risorse di elaborazione per eseguire applicazioni o archiviare dati nel modo tradizionale richiede tempo e soldi. Le organizzazioni dovrebbero altrimenti acquistare attrezzature tramite processi di approvvigionamento che possono richiedere mesi. Devono investire in spazi fisici, solitamente sale specializzate con alimentazione e raffreddamento. Una volta implementati i sistemi, hanno bisogno di professionisti per gestirli e mantenerli [8].

IaaS è molto conveniente per le aziende perché

- non hanno la necessità di procurarsi, configurare o gestire l'infrastruttura autonomamente
- disponibilità on demand delle risorse
- pagano solo per quello che utilizzano
- permettono di fare automaticamente lo scale up e lo scale down delle risorse in modo rapido, in base ai picchi di domanda
- non hanno single point of failure: l'infrastruttura cloud offre ridondanza e tolleranza di errore integrate, con carichi di lavoro distribuiti su più server e strutture.

PaaS

Per Platform as a Service, noto anche come PaaS, ci riferiamo a un tipo di modello di servizio di cloud computing che offre una piattaforma cloud flessibile e scalabile per sviluppare, eseguire il deployment, eseguire e gestire app. PaaS fornisce tutto ciò di cui gli sviluppatori hanno bisogno per lo sviluppo di applicazioni senza la fatica di aggiornare il sistema operativo e gli strumenti di sviluppo o di gestire l'hardware. L'intero ambiente PaaS, o piattaforma, è invece fornito da un provider di servizi di terze parti tramite il cloud. Le applicazioni vengono create direttamente nel sistema PaaS ed è possibile eseguirne immediatamente il deployment al completamento [9]. Dal punto di vista di un'azienda, il PaaS è vantaggioso perché

- si ha *Time to market* più rapido, non occorre creare una piattaforma di sviluppo delle applicazioni *ad hoc*
- serve poca manutenzione, il provider è responsabile di tenere tutto aggiornato
- consente di fare lo scale down per i periodi a basso traffico o di fare lo scale up per soddisfare tempestivamente picchi di richieste inaspettati.
- i team di sviluppo e DevOps possono accedere a servizi e strumenti PaaS condivisi da qualsiasi luogo

CaaS

Containers as a Service (CaaS) è un servizio cloud che aiuta a gestire e distribuire app utilizzando l'astrazione basata su container. CaaS può essere distribuito on premise o in un cloud. Il provider offre il framework, o la piattaforma di orchestrazione, sulla quale i container vengono distribuiti e gestiti. Grazie ai container è possibile creare ambienti omogenei dove sviluppare e distribuire applicazioni cloud native eseguibili su qualsiasi sistema [10].

Kubernetes è una piattaforma open source per l'orchestrazione dei container Linux, sviluppata in origine dagli ingegneri di Google, che consente di automatizzare lo sviluppo, la gestione e la scalabilità delle app, raggruppando in cluster i container in esecuzione su host Linux e automatizzandone la gestione. La maggior parte dei processi manuali coinvolti nel deployment e nella scalabilità delle app containerizzate viene gestita per conto dell'utente, dietro le quinte. Kubernetes offre le funzionalità di orchestrazione e gestione dei container necessarie per distribuire i container su larga scala, su più host server con numerosi livelli di sicurezza, gestendo l'integrità di tali container nel tempo.

Monitoring Query Language (MQL)

Google Cloud Platform fornisce un linguaggio per interrogare e analizzare i dati di monitoraggio chiamato Monitoring Query Language (MQL). Con MQL è possibile scrivere query per estrarre informazioni da metriche e registri di monitoraggio, così da poter effettuare analisi approfondite sulle prestazioni di servizi e delle risorse su GCP. Offre una sintassi intuitiva e potente per filtrare, aggregare e manipolare

i dati di monitoraggio, rendendo più facile comprendere e gestire l'infrastruttura cloud [11]. I passi fondamentali per creare una query MQL sono i seguenti:

1. Si specifica quale metrica si vuole analizzare attraverso un'endpoint e.g `run.googleapis.com/container/instance_count`
2. Si definisce un filtro per limitare i dati da analizzare, spesso si inserisce una regex. Possono essere basati su etichette e.g `region`, `project-id`
3. Si possono aggregare i dati utilizzando funzioni d'aggregazione e.g `sum`, `avg`
4. Si definisce l'allineamento dei dati, in modo che siano comparabili tra loro su intervalli regolari e.g `align delta(1h)`. In particolare:
   - applicando `delta(1h)` i valori rappresenteranno la differenza (assoluta) rispetto all'ora precedente.
   - applicando `rate(1h)` i valori rappresenteranno la variazione percentuale rispetto all'ora precedente.
5. Si specifica che intervallo di tempo utilizzare per campionare i dati e.g `every 1h`
6. Si possono raggruppare i dati per ottenere un unico valore per ciascun intervallo di tempo e.g `group_by [], [value_mean:mean(value)]`

```
fetch cloud_run_revision
    | metric 'run.googleapis.com/container/instance_count'
    | filter (resource.service_name =~ '^myServicePrefix-(.*)')
    | align rate(1h)
    | every 1h
    | group_by [], [value_instance_count_aggregate:
        aggregate(value.instance_count)]
```

Listing 2.1: Esempio di una query MQL

recupera i dati delle istanze di Cloud Run i cui servizi iniziano con il prefisso `myServicePrefix-`. Allinea quindi i dati a intervalli di un'ora e infine restituisce il conteggio medio complessivo delle istanze di tutti i *timeinterval* per quei servizi.

2.2 Java

E' un linguaggio di programmazione interpretato e orientato agli oggetti, sviluppato dalla Sun Microsystems nei primi anni 1990. Il codice sorgente di un programma scritto in Java non viene compilato in linguaggio macchina (dipendente dalla piattaforma) ma in un linguaggio intermedio, il java byte code, (indipendente dalla piattaforma) che per essere eseguito richiede solo l'uso di un interprete, la Java Virtual Machine.

2.3 Spring Boot

Spring Boot è un framework open-source del linguaggio Java che semplifica lo sviluppo di applicazioni complesse. Mira a semplificare il processo di creazione di applicazioni web robuste e scalabili. Si basa sul popolare framework Spring e fornisce un insieme di funzionalità che semplificano la configurazione, il test e l'esecuzione di applicazioni Spring. Quest'ultimo è un framework molto potente per lo sviluppo di applicazioni Java, tuttavia richiede molta configurazione. Spring Boot risolve questo problema automatizzando molte attività, rendendo l'esperienza di sviluppo semplice e veloce ma convervando la potenza di Spring.

Spring fa ampio uso del concetto di Bean e della dependency injection. La dependency injection (DI) è un processo in cui gli oggetti definiscono le loro dipendenze (cioè gli altri oggetti con cui lavorano) solo attraverso gli argomenti del costruttore, gli argomenti di un "factory method" o le proprietà impostate sull'istanza dell'oggetto dopo che è stato costruito o restituito da un "factory method". Il container inietta quindi queste dipendenze quando crea il bean. Il codice è più pulito e il disaccoppiamento è più efficace quando gli oggetti vengono forniti con le loro dipendenze. L'oggetto non cerca le sue dipendenze e non conosce la posizione o la classe delle dipendenze. Di conseguenza, le classi diventano più facili da testare, in particolare quando le dipendenze sono da interfacce o classi base astratte, che consentono di utilizzare implementazioni stub o mock negli unit test [12].
Le astrazioni di Spring Boot più utilizzate sono state:

@Component è uno stereotipo che indica a Spring di gestire una classe come un bean. Ciò significa che Spring:

- riconosce la classe all'avvio dell'applicazione
- instanzia il bean ogni volta che sarà necessario
- gestisce il ciclo di vita del bean, quindi la creazione, aggiornamento e distruzione dello stesso.

@Service è uno stereotipo che contrassegna specificamente una classe come componente responsabile di implementare la business logic dell'applicazione. E' una specializzazione del *@Component*.

Spring Boot JPA

Spring Boot JPA è una parte del framework Spring che semplifica lo sviluppo di applicazioni Java che utilizzano la *Java Persistence API* (JPA) per l'accesso ai dati. La JPA non è altro che una specifica Java per l'accesso, il mantenimento e la gestione dei dati tra gli oggetti Java e un database relazionale. Fornisce un insieme di interfacce e annotazioni per la mappatura di oggetti Java su tabelle di database e viceversa. I punti fondamentali di Spring Boot JPA sono:

Entity Mapping semplifica la mappatura tra *PoJo* e le tabelle del database. Ciò si ottiene attraverso annotazioni - e.g `@Entity`, `@Table`, `@Column` - che definiscono come gli oggetti Java vengono memorizzati nel database.

Repository fornisce l'interfaccia CrudRepository. Questa interfaccia fornisce operazioni `CRUD` (Create, Read, Update, Delete) di base per le entità.

Configurazione Spring configura automaticamente JPA cercando le entità e i repository nel classpath dell'applicazione. Inoltre è facilmente configurabile attraverso l'`application.yml`.

Capitolo 3

Progettazione e sviluppo

3.1 Requisiti del connettore ricevuti

Il connettore è un Cloud Run Job pianificato per essere eseguito all'01:00 di ogni notte. Quello che fa è raccogliere e inviare metriche dei servizi descritti nel file di configurazione JSON. Viene monitorata ogni ora del giorno precedente all'esecuzione del job.

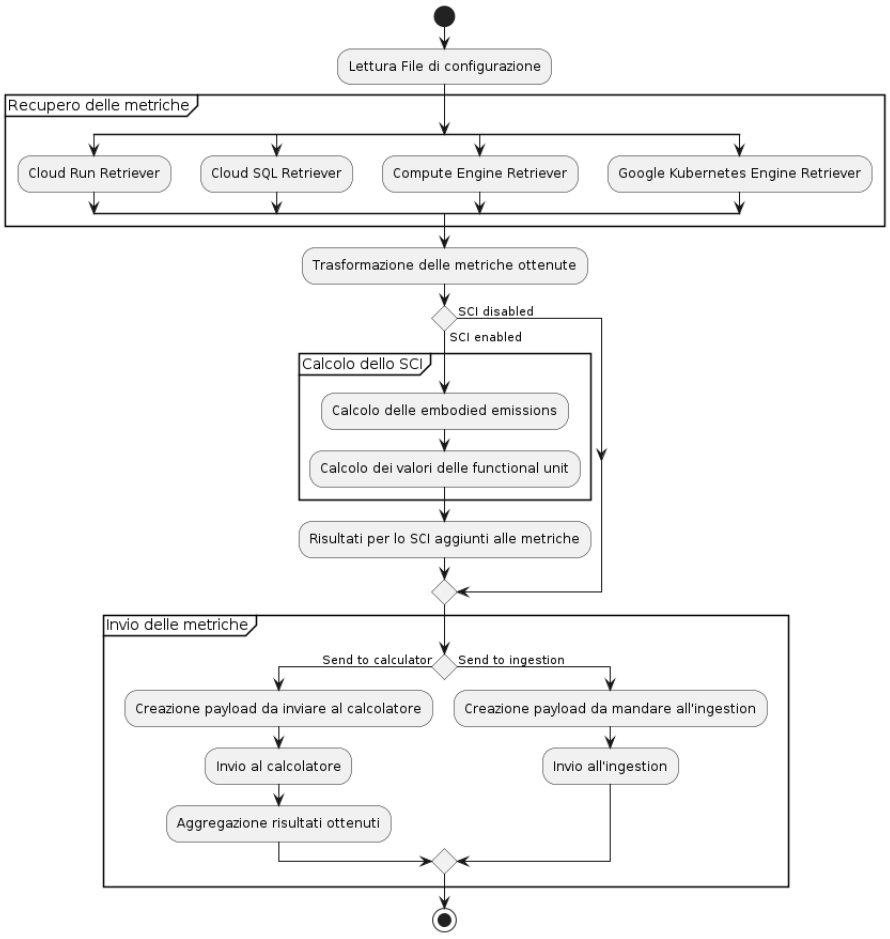


Figura 3.1: Diagramma delle attività

3.1.1 File di configurazione

Il file di configurazione è un file JSON in cui sono specificati i servizi da monitorare e informazioni utili per consentire il monitoraggio.

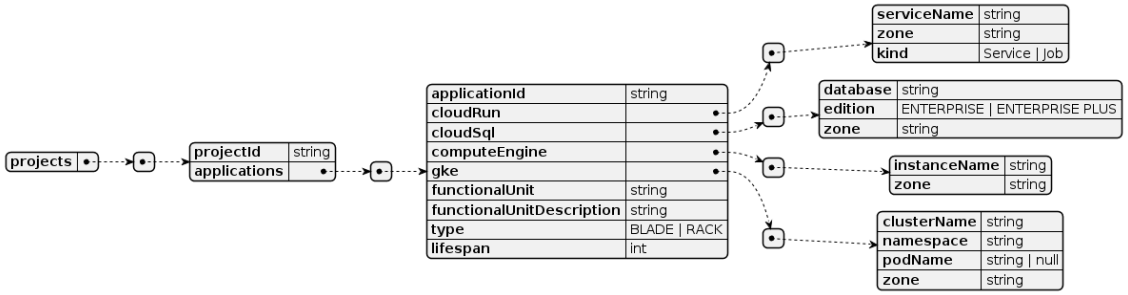


Figura 3.2: Schema JSON di configurazione

- Per ogni progetto vengono specificate delle *applications*, gruppi logici di componenti di GCP da monitorare.
  - Per ogni applicazione si specificano i vari servizi da monitorare (e.g tutti i cloudRun e i cloudSql di interesse) e i parametri necessari per ottenere lo SCI, qual ora fosse stato abilitato.

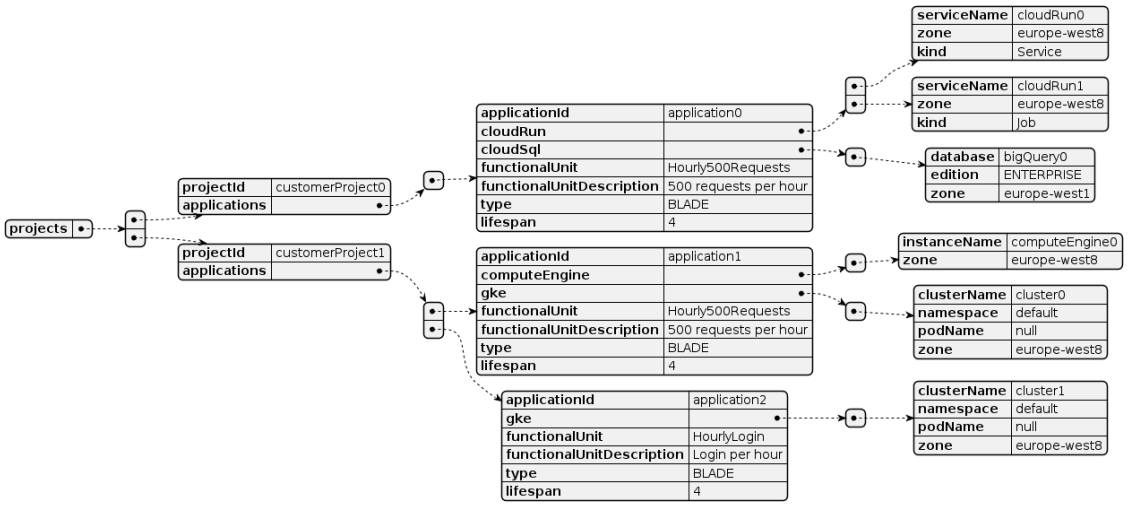


Figura 3.3: Esempio JSON di configurazione

3.1.2 Servizi monitorabili

Al momento il connettore è in grado di monitorare quattro tipologie di servizi diversi:

Servizio	Modello di servizio
Cloud Run	PaaS
Cloud SQL	IaaS
Compute Engine	IaaS
Google Kubernetes Engine (GKE)	CaaS

Cloud Run

Cloud Run è una piattaforma di computing che consente di eseguire container direttamente sull'infrastruttura scalabile di Google. E' possibile eseguire il deployment di codice scritto in qualsiasi linguaggio di programmazione, a patto di fornire un'immagine container. In Cloud Run, il codice può essere eseguito come service o come job:

- un service permette di eseguire codice che risponda a richieste web o eventi.
- un job permette di eseguire codice che esegue un "lavoro" e viene chiuso quando il lavoro è terminato.

Data la natura container-based di Cloud Run, le **metriche** interessanti per il calcolo della CO2 sono:

CPU Utilization Quanta CPU viene utilizzata da tutti i container rispetto alla CPU totale allocata.

RAM Size Quantità media di RAM allocata per container moltiplicata per il numero di istanze attive in quel momento

Number of Virtual Cores Quantità media di virtual core allocati per container moltiplicata per il numero di istanze attive in quel momento

Cloud SQL

Cloud SQL è un servizio di database relazionale completamente gestito per MySQL, PostgreSQL e SQL Server. Molto comoda se si usano Cloud Run Jobs/Services o istanze Compute Engine, perché l'ecosistema di GCP consente la connessione diretta alle istanze di Cloud SQL all'interno della stessa rete VPC (**Virtual Private Cloud**). Ogni istanza Cloud SQL è alimentata da una macchina virtuale (VM) in esecuzione su un server Google Cloud host. Ogni VM gestisce il programma di database, ad esempio MySQL Server, PostgreSQL o SQL Server, e agenti di servizio che forniscono servizi di supporto, come il logging e il monitoraggio. Cloud SQL non è da sottovalutare come emissioni di CO2, le metriche interessanti per il calcolo della CO2 sono:

CPU Reserved Cores Numero minimo di core riservati per il database. Maggiore è il numero di core riservati, maggiore sarà il consumo energetico e le emissioni di CO2.

CPU Usage Percentuale di tempo in cui la CPU viene effettivamente utilizzata. Picchi di CPU Usage corrispondono a momenti della giornata particolarmente intensi.

CPU Type Tipo di CPU utilizzata dal database, diversi tipi di CPU hanno diverse efficienze energetiche.

Disk Type Tipo di disco utilizzato per l'archiviazione dei dati nel database. Tipicamente la scelta è fra SSD o HDD, dove i primi consumano meno energia dei secondi.

RAM Size Quantità di RAM allocata al database, il cui consumo energetico non è irrilevante.

Compute Engine

Compute Engine è un servizio di computing e hosting che consente di creare ed eseguire macchine virtuali sull'infrastruttura di Google. Compute Engine offre scalabilità, prestazioni e valore che ti consentono di avviare facilmente cluster di computing di grandi dimensioni sull'infrastruttura di Google.

Le istanze di Compute Engine possono:

- eseguire le immagini pubbliche per Linux e Windows Server fornite da Google, nonché le immagini personalizzate private che puoi creare o importare dai sistemi esistenti.

- eseguire il deployment dei container Docker.

La virtualizzazione di Compute Engine ha un impatto notevole nei consumi elettrici, le metriche interessanti per il calcolo delle emissioni sono:

CPU Usage Percentuale di tempo in cui la CPU viene effettivamente usata dalla VM

Disk Size Dimensione dello spazio di archiviazione allocato alla VM

CPU Platform Famiglia di CPU utilizzata dalla VM

Reserved Cores Numero minimo di core riservati per la VM

RAM Size Quantità di RAM allocata alla VM

Google Kubernetes Engine (GKE)

Google Kubernetes Engine (GKE) è un servizio per l'esecuzione, la gestione e l'orchestrazione di container su GCP. Si basa su Kubernetes, una piattaforma open source per l'automazione della distribuzione, della gestione, del ridimensionamento e della rete di applicazioni containerizzate. GKE è un Container as a Service (CaaS), perché offre un modo semplificato per eseguire container senza dover gestire l'infrastruttura sottostante. Con GKE, Google si occupa della gestione dei nodi del cluster, del networking, del load balancing e di altre attività infrastrutturali. Essendo GKE una piattaforma di orchestrazione di container, è il servizio di cui teniamo in considerazione più metriche. Questo perché ha molti fattori che influenzano l'impatto ambientale, tra cui:

CPU Limit Cores Numero massimo di core che un container può utilizzare.

CPU Limit Utilization Percentuale massima di utilizzo della CPU per container.

Memory Limit Quantità massima di RAM che un pod può utilizzare.

CPU Machine Type Famiglia di CPU utilizzata dai nodi del cluster GKE e.g Intel Xeon.

Minimum CPU Type Il tipo di CPU su cui viene eseguito il container.

Disk type Tipo di disco utilizzato per l'archiviazione dei dati dei pod.

3.1.3 Ottenere le metriche dei servizi

E' stato fondamentale per l'implementazione del recupero delle metriche, perché mi ha permesso di fare verifiche sui dati ancora prima di scrivere codice java; questo mi ha portato a fare delle correzioni sulle specifiche funzionali che mi avevano fornito, accorciando i tempi di messa in produzione del connettore. Quasi ogni metrica ha la sua corrispettiva query MQL, alcune invece sono approssimazioni fatte su alcune risorse delle macchine oppure dati statici.

Cloud Run

E' stato possibile ottenere solo la CPU Utilization tramite MQL:

```
1 fetch cloud_run_revision
2  | metric 'run.googleapis.com/container/cpu/utilizations'
3  | filter (resource.service_name =~ '^myservice-prefix-(.*)')
4  | align delta(1h)
5  | every 1h
6  | group_by [], [value_utilizations_mean:
7        mean(value.utilizations)]
```

Listing 3.1: Query MQL per la CPU Utilization per Cloud Run

invece, sia RAM Size che il numero di virtual core sono stati ottenuti dalla seguente considerazione:

> Un Cloud Run può avere più istanze per gestire carichi di lavoro variabili e garantire una maggiore scalabilità. Le istanze di Cloud Run eseguono i container, che sono pacchetti software autocontenuti che includono l'applicazione e tutte le sue dipendenze. Quando viene avviato un container Cloud Run, il sistema assegna automaticamente risorse CPU e memoria in base alle esigenze dell'applicazione.

Possiamo calcolare, per una singola istanza, il numero di core virtual core e la RAM attraverso le medie

$$\texttt{vCoresAvg} = \frac{\sum_{i=1}^{n} containers[i].vCores}{\#containers} \qquad \texttt{memoryAvg} = \frac{\sum_{i=1}^{n} containers[i].memory}{\#containers}$$

Una volta ottenuto il numero di istanze di un Cloud Run, è possibile approssimare il numero totale di virtual core e la RAM attraverso una semplice moltiplicazione:

$$\texttt{vCores} = \texttt{vCoresAvg} \cdot \texttt{instanceCount} \qquad \texttt{memory} = \texttt{memoryAvg} \cdot \texttt{instanceCount}$$

Per quanto riguarda l'instance count, è stato sufficiente implementare la seguente query MQL:

```
fetch cloud_run_revision
 | metric 'run.googleapis.com/container/instance_count'
 | filter (resource.service_name =~ '^myservice-prefix-(.*)')
 | align rate(1h)
 | every 1h
 | group_by [], [value_instance_count_aggregate:
      aggregate(value.instance_count)]
```

Listing 3.2: Query MQL per l'instance count per Cloud Run

Cloud SQL

La CPU Usage viene recuperata implementando in java la seguente query MQL:

```
    fetch cloudsql_database
      | metric 'cloudsql.googleapis.com/database/cpu/usage_time'
      | filter (resource.database_id == 'myDatabaseId')
      | align delta(1h)
      | every 1h
      | group_by [], [value_usage_time_mean: mean(value.usage_time)]
```

Listing 3.3: Query MQL per la CPU Usage di Cloud SQL

il resto delle metriche da recuperare non variano col tempo, vengono recuperate direttamente dall'istanza.

- `CPU Reserved Cores` e `RAM Size` vengono ottenuti dalla stringa `Tier`[1] che segue questo pattern:

  `TIER = 'db-custom-{reservedCores}-{ramInMb}'`

  in Java viene ottenuta dall'oggetto `Settings` proveniente dalla classe statica `DatabaseInstance`.

- `Disk Type` in Java viene ottenuta dal metodo `getDataDiskType()` fornito dalla classe statica `DatabaseInstance`.

- `Disk Size` in Java viene ottenuta dal metodo `getDataDiskSizeGb()` fornito dalla classe statica `DatabaseInstance`.

- `CPU Type` viene determinato in base alla *database edition*[2]:
  - se `ENTERPRISE` allora il `CPU Type` sarà una `e2-average-cpu`
  - se `ENTERPRISE PLUS` allora il `CPU Type` sarà una `c3-average-cpu`

Compute Engine

CPU Reserved Cores, RAM Size e CPU Usage vengono ottenuti dalle seguenti query MQL:

```
fetch gce_instance
    | metric 'compute.googleapis.com/instance/cpu/reserved_cores'
    | filter (metric.instance_name == 'myInstance')
    | align rate(1h)
    | every 1h
    | group_by [], [value_usage_time_mean: mean(value.usage_time)]
```

Listing 3.4: Query MQL per i CPU Reserved Cores di Compute Engine

```
fetch gce_instance
    | metric 'compute.googleapis.com/instance/memory/balloon/ram_size'
    | filter (metric.instance_name == 'myInstance')
    | align rate(1h)
    | every 1h
    | group_by [], [value_ram_size_mean: mean(value.ram_size)]
```

Listing 3.5: Query MQL per la RAM Size di Compute Engine

```
fetch gce_instance
    | metric 'compute.googleapis.com/instance/cpu/usage_time'
    | filter (metric.instance_name == 'myInstance')
    | align delta(1h)
```

[1]Definisce il tipo di macchina sottostante assegnata al database. Determina le risorse di calcolo, come CPU e memoria, disponibili per l'istanza del database.

[2]Si riferisce a un livello di servizio che determina il set di funzionalità, le capacità di prestazione e i prezzi.

```
5       | every 1h
6       | group_by [], [value_usage_time_mean: mean(value.usage_time)]
```

Listing 3.6: Query MQL per la CPU Usage di Compute Engine

per quanto riguarda la CPU Usage, la metrica `instance/cpu/usage_time`:

> Delta vCPU usage for all vCPUs, in vCPU-seconds. To compute the per-vCPU utilization fraction, divide this value by $(end - start) \cdot n$, where end and start define this value's time interval and n is `CPU Reserved Cores` at the end of the interval [13].

quindi per ottenere una percentuale, sarà necessario dividere la CPU Usage per 3600 (perché monitoriamo ogni ora) e i CPU Reserved Cores ottenuti precedentemente:

$$\texttt{CPU\_Usage} = \frac{\texttt{CPU\_Usage}}{3600 \cdot \texttt{CPU\_Reserved\_Cores}}$$

Disk Size e CPU Platform vengono ottenuti dall'istanza usando l'`InstanceClient`.

Google Kubernetes Engine

Le metriche Machine Type, Minimum CPU Type e Disk Type vengono ottenuti da una `GetClusterRequest` effettuata attraverso il `ClusterManagerClient`. Per quanto riguarda il resto delle metriche, vengono utilizzate le seguenti query MQL:

```
1  fetch k8s_container
2      | metric 'kubernetes.io/container/cpu/limit_cores'
3      | filter (resource.cluster_name == 'myCluster')
4      | group_by 1h, [value_limit_cores_mean: mean(value.limit_cores)]
5      | every 1h
6      | group_by [resource.project_id],
7          [value_limit_cores_mean_aggregate: aggregate(
       value_limit_cores_mean)]
```

Listing 3.7: Query MQL per la CPU Limit Cores di GKE

```
1  fetch k8s_container
2      | metric 'kubernetes.io/container/cpu/limit_utilization'
3      | filter (resource.cluster_name == 'myCluster')
4      | group_by 1h, [value_request_utilization_mean: mean(value.
       limit_utilization)]
5      | every 1h
6      | group_by [], [value_request_utilization_mean_aggregate:
7          aggregate(value_request_utilization_mean)]
```

Listing 3.8: Query MQL per la CPU Limit Utilization di GKE

```
1  fetch k8s_container
2      | metric 'kubernetes.io/container/memory/limit_bytes'
3      | filter (resource.cluster_name == 'myCluster')
```

```
4       | group_by 1h, [value_request_bytes_mean: mean(value.request_bytes)
        ]
5       | every 1h
6       | group_by [], [value_request_bytes_mean_aggregate:
7           ggregate(value_request_bytes_mean)]
```

Listing 3.9: Query MQL per il Memory Limit di GKE

3.2 Architettura del connettore

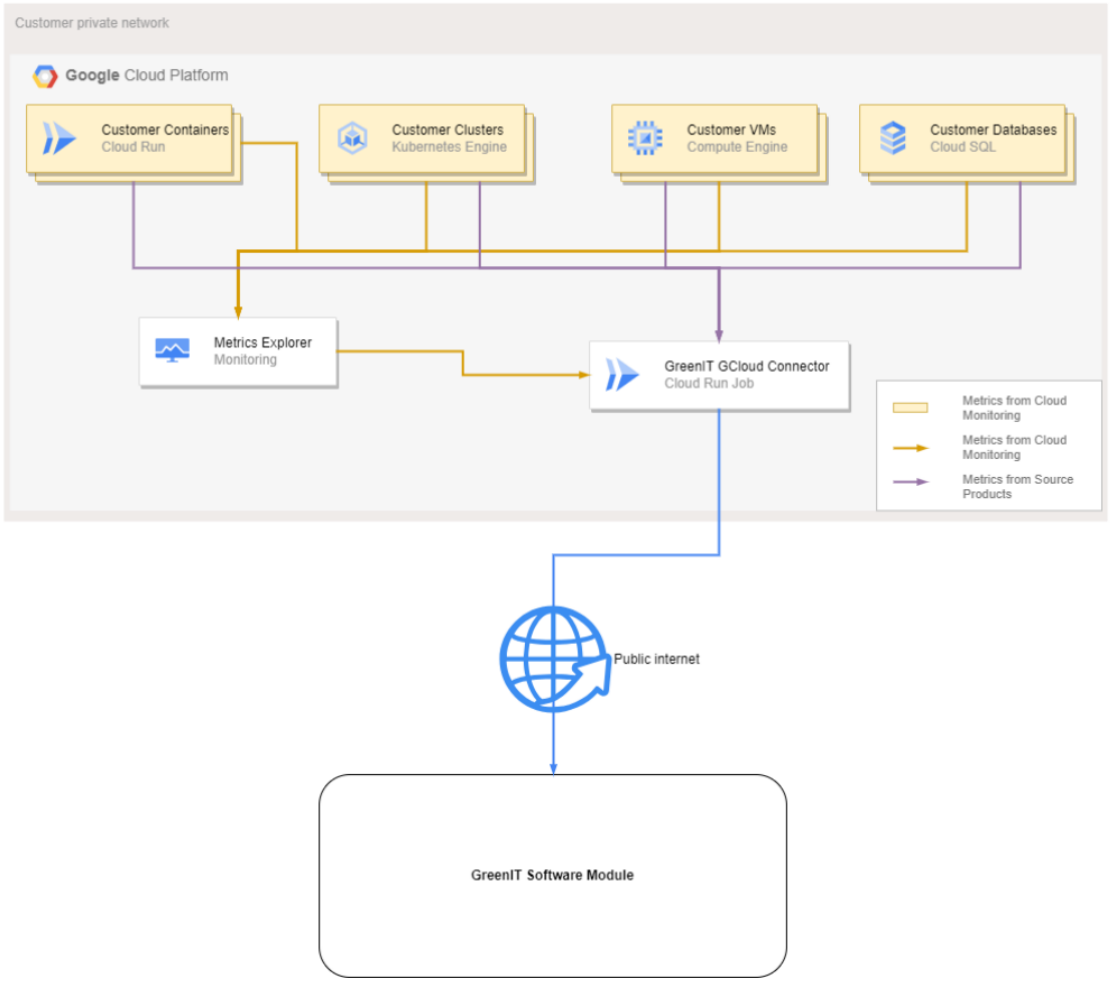


Figura 3.4: Diagramma delle classi Retriever

Nell'ambiente Google Cloud Platform, un cliente dispone di diverse risorse come container, cluster, macchine virtuali e database. Il servizio Metrics Explorer di Google Cloud Monitoring raccoglie le metriche di monitoraggio da queste risorse. Il connettore, implementato come Cloud Run Job, funge da ponte tra l'ambiente Google Cloud e Green IT. Esso raccoglie le metriche provenienti da due fonti diverse:

1. Metriche da Cloud Monitoring (metriche di risorse come CPU, memoria, traffico di rete, etc.)

2. Metriche dai prodotti sorgente (potenzialmente altre metriche specifiche dell'applicazione)

Il connettore invia quindi questi dati di monitoraggio raccolti attraverso Internet pubblico al software Green IT.

3.3 Scelte di progettazione

3.3.1 Memorizzazione del file di configurazione

Il file di configurazione è un JSON caricato all'interno di un bucket di Google Cloud Storage. E' stato scelto JSON come formato perché:

- è un formato con una sintassi minimalista, leggero e facile da leggere e scrivere per l'uomo.
- è diventato il formato standard per lo scambio di dato nelle API.
- dati in JSON possono essere analizzati e generati in modo efficiente

In più, mantenere il file su Google Cloud Storage:

- elimina la necessità di gestire diverse copie del file in ambienti separati
- offre un sistema di controllo delle versioni, che consente di tenere traccia delle modifiche apportate al file JSON nel tempo
- offre una client library per Java, che rende molto semplice l'integrazione con il connettore.

3.3.2 Profili di Spring

Il connettore supporta due diverse modalità: `sci_enabled` e `sci_disabled`. Se il calcolo dello SCI è abilitato, vengono istanziati molti più componenti Spring - come vedremo più avanti - che andrebberò ad appesantire inutilmente l'esecuzione del connettore quando il calcolo dello SCI è disabilitato. Per evitare scenari di questo tipo sono stati scelti gli *Spring Profiles*, forniscono un modo per segregare parti della configurazione dell'applicazione e renderle disponibili solo in determinati *environments* [14].

3.3.3 Serializzazione e deserializzazione di oggetti

La serializzazione e la deserializzazione sono processi fondamentali nel contesto dello scambio e della persistenza dei dati. Comportano la conversione di oggetti di dati in un formato adatto alla memorizzazione o alla trasmissione e viceversa.

- La **serializzazione** è il processo di conversione dello stato di un oggetto in una sequenza di byte (una stringa) che può essere trasmesso in rete o salvato in un file o in un database.

- La **deserializzazione** è il processo inverso della serializzazione. Consiste nel prendere i dati serializzati e ricostruire l'oggetto o gli oggetti originali a partire da essi.

Entrambi i processi di serializzazione e deserializzazione devono seguire regole o formati predefiniti per garantire che i dati possano essere ricostruiti con precisione. Esistono diversi formati standardizzati, ma i più famosi e utilizzati ad oggi rimangono XML e JSON. Le metriche ottenute da Google Cloud Platform devono essere formattate, aggregate e serializzate in un certo modo prima di essere mandate al calcolatore. Per raggiungere questo obiettivo in maniera semplice si è optato per la libreria `Jackson`, che offre una suite di strumenti per l'elaborazione dei dati per Java, tra cui il parser/generatore di JSON, la libreria per il matching data-binding (POJO da e verso JSON) [15]. La serializzazione e la deserializzazione sono due operazioni fondamentali all'interno del connettore, per semplicità sono stati definiti molte classi Data Transfer Object. Quest'ultime facilitano il trasporto di dati tra i processi del connettore e la serializzazione per l'invio al calcolatore.

3.3.4 Logging

Il *logging* è una funzionalità che permette, durante il funzionamento di una nostra applicazione, di inviare messaggi ad una determinata destinazione che tenga traccia di quanto sta avvenendo. Nei messaggi, in genere, si inseriscono informazioni concernenti possibili errori verificatisi o dati di carattere statistico o puramente informativo su quello che il programma fa. Questa funzionalità è essenziale nel connettore, viene implementata attraverso la `Default Logback Logging` fornita da Spring Boot e facilitata da *logging facade* SLF4J (Simple Logging Facade for Java). SLF4J non è un framework per il logging bensì è una facade che semplifica il compito del logging al programmatore. Utilizzarlo fornisce diversi vantaggi:

- Le applicazioni non sono vincolate a uno specifico framework per il logging, riducendo il rischio di vendor lock-in.
- Fornisce un insieme di metodi (e.g `debug`, `info`, `warn`, `error`), per scrivere log in modo uniforme su tutta la codebase, indipendentemente dal framework di logging sottostante.

3.4 Implementazione del connettore

In questa sezione verranno trattati gli aspetti più tecnici della progettazione del connettore, che comprendono:

- i componenti `Retriever` per ottenere le metriche
- l'implementazione - non banale - dello SCI in dettaglio
- la creazione dei request body tramite factory
- l'invio dei dati al calcolatore

3.4.1 Variabili d'ambiente e configurazioni

Il connettore prevede delle variabili d'ambiente, fondamentali per personalizzare l'esecuzione dell'adapter. Di seguito riporto una breve descrizione di ciascuno di essi:

`SPRING_PROFILES_ACTIVE` permette di selezionare l'`application-%profile.yml` con cui eseguire l'applicazione; modificherà il comportamento dell'adapter ed eviterà inutili istanziazioni di bean.

`APIKEY` fondamentale per autorizzare le chiamate HTTP effettuate dall'adapter verso l'apigateway

`BUCKET_NAME` nome del bucket gcloud in cui si trova il file di configurazione

`CONFIG_FILE_NAME` nome del file di configurazione

`SEND_TO_CALCULATOR` variabile booleana

- se `true` il connettore invia una POST al calcolatore con le metriche ottenute, nessun dato viene salvato sul database (creato per esigenze di test)
- se `false` il connettore invia le metriche ottenute all'endpoint `v2/ingestion-event`, i dati vengono salvati sul database

`DB_URL` indirizzo IP del database MySQL da cui calcolare le functional units.

`DB_PORT` porta in cui è in ascolto MySQL (di default è 3306).

`DB_NAME` nome del database MySQL.

`DB_USER` username per autenticare Spring JPA al database mysql.

`DB_PASSWORD` password per autenticare Spring JPA al database mysql.

E' stata definita una classe `AdapterConfigProperties`. Fornisce un modo strutturato per contenere e gestire le variabili d'ambiente rilevanti dell'adapter, facilitando l'accesso di queste proprietà in tutto il codice.

```
1 @ConfigurationProperties("adapter")
2 public record AdapterConfigProperties(
3         String bucketName,
4         String configFileName,
5         String apikey,
6         boolean sendToCalculator,
7         boolean sciEnabled
8 ) {;}
```

- è un Java Record, modo conciso per dichiarare classi che trasportano dati. I record generano automaticamente getter per tutti i loro componenti.

- l'annotazione `@ConfigurationProperties("adapter")` indica a Spring Boot che questa classe è responsabile della gestione di un sottoinsieme di proprietà che iniziano con `adapter` nel file `application.yml` 5. Nello specifico mappa le proprietà ai campi del record.

Se un componente di Spring ha bisogno di accedere alle variabili d'ambiente, è sufficiente iniettare in esso un'istanza di `AdapterConfigProperties`.

Ottimizzazione dei profili

Sono stati definiti due profili spring, entrambi ereditano le variabili d'ambiente definite nell'`application.yml` 5.

sci_enabled risiede nel file `application-sci_enabled`

```
spring:
  datasource:
    url: jdbc:mysql://${DB_URL:localhost}:${DB_PORT:3306}/${DB_NAME}
    username: ${DB_USER}
    password: ${DB_PASSWORD}
  jpa:
    hibernate:
      ddl-auto: none
      naming:
        physical-strategy: org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
    show-sql: true
adapter:
  sciEnabled: true
```

definisce a quale URL e porta stabilire la connessine in modo tale da interagire col database MySQL. Il nome utente e la password sono utilizzati per l'autenticazione. Hibernate è configurato per non modificare lo schema del database, limitando i permessi del connettore in sola lettura (*read-only*). Abilita la variabile d'ambiente `sciEnabled`.

sci_disabled risiede nel file `application-sci_disabled`

```
spring:
  autoconfigure:
    exclude:
      - org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration
      - org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
adapter:
  sciEnabled: false
```

Escludendo `HibernateJpaAutoConfiguration`, l'adapter rinuncia esplicitamente alla configurazione automatica di Hibernate JPA fornita da Spring Boot. Quindi non configurerà automaticamente Hibernate JPA per le funzionalità ORM (Object-Relational Mapping).

L'esclusione di `DataSourceAutoConfiguration` impedisce a Spring Boot di configurare automaticamente i bean per la connettività al database.

In sostanza il profilo `sci_disabled` evita sprechi di risorse, l'adapter rinuncia esplicitamente alla configurazione automatica:

- di Hibernate JPA per le funzionalità ORM (Object-Relational Mapping), scansione delle entità, l'impostazione della connessione al database e la gestione delle transazioni
- dei bean per la connettività al database

cose che sarebbero state utili solo in caso di monitoraggio dello SCI.

3.4.2 CommandLineRunner

Il core principale del connettore è la classe `BootStrap`, che implementa l'interfaccia `CommandLineRunner`[3] e fa override del metodo `run`. Il codice scritto all'interno di `run` viene eseguito all'avvio dell'applicazione, visto che il connettore deve eseguire una routine ben delineata, il flusso di esecuzione parte con `run` e termina con lo stesso.

```
1  public class BootStrapData implements CommandLineRunner {
2      private final FootprintService footprint;
3      private final AdapterConfigProperties adapterConfig;
4      private FunctionalUnitSingletonRetriever
       functionalUnitSingletonRetriever;
5
6      @Autowired
7      public BootStrapData(FootprintServiceImpl footprint,
       AdapterConfigProperties adapterConfig) {
8          this.footprint = footprint;
9          this.adapterConfig = adapterConfig;
10     }
11
12     @Autowired(required = false)
13     public void setFunctionalUnitSingletonRetriever(
       FunctionalUnitSingletonRetriever functionalUnitSingletonRetriever) {
14         this.functionalUnitSingletonRetriever =
       functionalUnitSingletonRetriever;
15     }
16
```

[3]Interfaccia utilizzata per indicare che un bean deve essere eseguito quando è contenuto in una SpringApplication.

```
17      @Override
18      public void run(String... args) throws Exception
19  }
```

Vengono iniettate tre classi:

1. `FootprintService` tramite *constructor injection*, questa istanza verrà utilizzata per eseguire operazioni quali *POST* delle metriche e il calcolo delle emissioni di CO2.

2. `AdapterConfigProperties` tramite *constructor injection*, una classe di configurazione che contiene le variabili d'ambiente relative al comportamento dell'adapter.

3. `FunctionalUnitSingletonRetriever` una classe singleton responsabile della gestione dei Functional Unit Retriever. Viene iniettato tramite setter. Il motivo per cui si usa un *setter injection* piuttosto che un costruttore injection è quello di renderlo opzionale (`@Autowired(required = false)`), consentendo all'applicazione di funzionare anche se il bean `FunctionalUnitSingleton-Retriever` non è disponibile (quando lo SCI non viene monitorato).

Il flusso del metodo `run` è diviso logicamente in 3 fasi:

1. Fase di lettura del file di configurazione e creazione delle strutture dati che ospiteranno i risultati.

2. Fase di ottenimento delle metriche e scrittura delle stesse in un DTO.

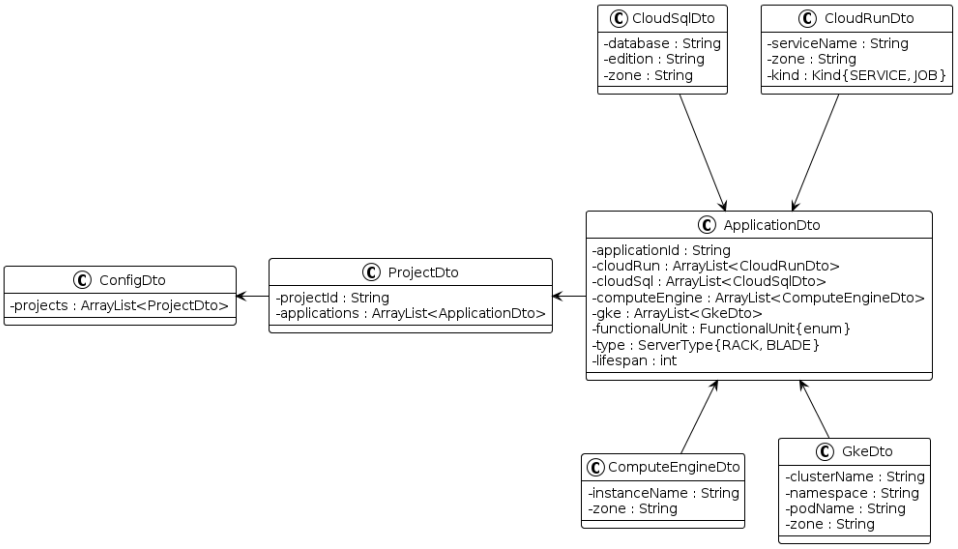
3. Fase di invio delle metriche.

3.4.3 Data Transfer Object (DTO)

Il Data Transfer Object è un design pattern comune nel contesto della programmazione orientata agli oggetti e il web development. Un DTO è un oggetto che trasporta dati tra i processi di un'applicazione, spesso viene utilizzato per incapsulare e trasferire dati tra diverse parti dello sistema (oppure tra diversi layer dell'applicazione, e.g tra un client e un server).

Per questo connettore ne sono stati definiti molteplici, io mi limiterò solo a elencare le macro categorie sinteticamente, per poi entrare in dettaglio di quelli fondamentali e meno banali.

DTO di configurazione risiedono all'interno del package `dto.configuration`, questi DTO vengono impiegati per effettuare la traduzione - *mapping* - del file di configurazione in un PoJo (Plain old Java object) di tipo `ConfigDto`. Ciascun progetto all'interno di `ConfigDto` rappresenta le specifiche di monitoraggio di un progetto da monitorare creato su Google Cloud Platform e.g i suoi cloud run, cluster di kubernetes...



DTO delle metriche ottenute risiedono all'interno del package `dto.metrics`, questi DTO vengono utilizzati per salvare le metriche ottenute tramite i Retrievers. Le classi peculiari di questa categoria sono `CloudRunResponseDto`, `CloudSqlResponseDto`, `ComputeEngineResponseDto` e `GkeResponseDto`.

Ogn'una di queste classi:

- possiede degli attributi per le metriche statiche, cioè quelle che non variano nel corso della giornata e.g `embodiedEmission`
- possiede un attributo `interval`, lista di 24 `ResultIntervalDto` che contiene le metriche variabili per ciascun intervallo di tempo.

DTO per il calcolo delle emissioni risiedono all'interno del package `dto.footprint`, questi DTO vengono utilizzati dalle `Factory` per creare i request body per le chiamate al calcolatore.

3.4.4 Retrievers

Per ottenere le metriche, sono state definite delle classi *Retriever*, le quali implementano l'interfaccia `DataRetriever` e sono specializzate nell'ottenere le metriche per ogni tipologia di servizio.

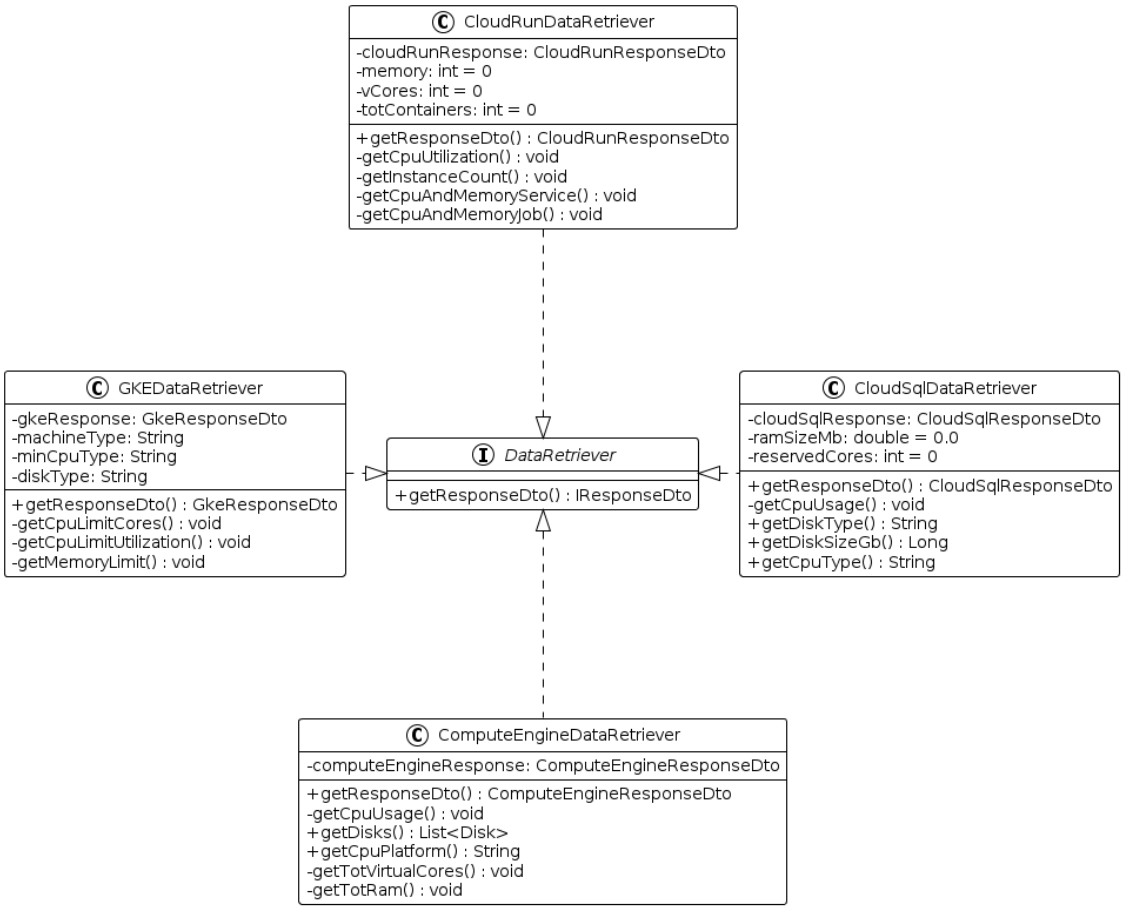


Figura 3.5: Diagramma delle classi Retriever

Come elaborano metriche orarie

Ciascuna classe ha definito all'interno dei metodi privati per l'acquisizione delle metriche orarie e.g `getCpuUtilization()`.

1. Creare una `ListTimeSeriesRequest` che soddisfi le specifiche della query MQL a cui fa riferimento. Per fare ciò bisogna definire un filtro e un'aggregazione:

```
1 String filter = String.format("metric.type = \"run.googleapis.com/
      container/cpu/utilizations\" AND resource.labels.service_name :
       \"%s\"" , serviceName);
2 Aggregation aggregation = Aggregation.newBuilder()
3     .setAlignmentPeriod(Duration.newBuilder().setSeconds(
      AGGREGATION_ALIGNMENT_PERIOD).build())
```

```
4       .setPerSeriesAligner(Aggregation.Aligner.ALIGN_DELTA)
5       .setCrossSeriesReducer(Aggregation.Reducer.REDUCE_MEAN)
6       .build();
```

Poi utilizzarli come parametri per la richiesta:

```
1  ListTimeSeriesRequest request = ListTimeSeriesRequest.newBuilder()
2          .setName(projectName.toString())
3          .setFilter(filter)
4          .setInterval(timeInterval)
5          .setAggregation(aggregation)
6          .setView(ListTimeSeriesRequest.TimeSeriesView.FULL)
7          .build();
```

2. Utilizzare il `MetricServiceClient` per mandare la richiesta. La risposta sarà un oggetto `ListTimeSeriesPagedResponse`, che conterrà le timeseries richieste

```
1  try (final MetricServiceClient metricServiceClient =
       MetricServiceClient.create()) {
2      ListTimeSeriesPagedResponse response = metricServiceClient
3              .listTimeSeries(request);
4      // code
5  }
```

3. Inserire il valore ottenuto da ogni timeserie nell'oggetto risposta della classe, operando attraverso il riferimento del risultato:

```
1  for (TimeSeries timeSeries : response.iterateAll()) {
2      int index = (timeSeries.getStartTimeSec() / 3600) % 24;
3      double value = timeSeries.getValue().getDoubleValue();
4      cloudRunResponse.getInterval().get(index).getResult().
       setCpuUtilization(value);
5  }
```

l'interval di ogni oggetto `Response` è un'arraylist inizializzato a 24 elementi. Se la metrica è assente in una determinata ora - e.g macchina spenta - allora l'elemento `ResultInterval` dell'arraylist `interval` rimarrà `null`. Come vedremo più avanti, questo semplifica la creazione delle richieste al calcolatore.

Come elaborano metriche statiche

Per *metrica statica* intendiamo una metrica di un componente che siamo sicuri non cambiare nell'arco della giornata, ad esempio:

- l'`Instance Count` non è una metrica statica. Infatti, quando un cloud run è spento l'`Instance Count` scende a zero; altrimenti, se attivo, è almeno pari a uno.

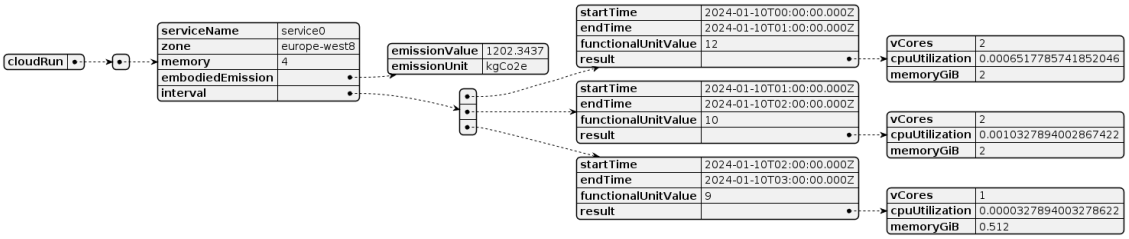


Figura 3.6: Oggetto `CloudRunResponseDto` dopo l'ottenimento delle metriche

- il `Disk Type` di un istanza Cloud SQL è una metrica statica. Infatti, durante la creazione dell'istanza viene specificata la tipologia di disco da utilizzare (SSD o HDD), e questa scelta non viene mai più modificata.

metriche del genere vengono recuperate attraverso delle semplici chiamate a delle classi fornite dalla Google Monitoring SDK per Java.

Per evitare inutili ripetizioni, farò un esempio generale

1. Viene ottenuta l'istanza del database identificato dal projectId e dal databaseId utilizzando la classe SQLAdmin:

```
SQLAdmin sqlAdminService = new SQLAdmin.Builder()
        .setApplicationName("GreenIT-adapter")
        .build();
DatabaseInstance databaseInstance = sqlAdminService.instances().
    get(projectId, databaseId).execute();
```

2. La `databaseInstance` possiede i metodi necessari per ottenere tutte le metriche statiche che servono:

```
String tier = databaseInstance.getSettings().getTier();
String diskType = databaseInstance.getSettings().getDataDiskType()
    ;
Long diskSizeGb = databaseInstance.getSettings().getDataDiskSizeGb
    ();
```

questo processo viene fatto da ogni retriever, l'unica cosa che cambia sono le classi con cui viene ottenuta l'istanza e.g per Cloud SQL è proprio `SQLAdmin` mentre per GKE è il `ClusterManagerClient`.

3.4.5 SCI

Proposta funzionale

Lo SCI (Software Carbon Intensity) permette di misurare la quantità di CO2 emessa globalmente dal software in relazione a quanto viene utilizzato. La formula per il calcolo dello SCI è la seguente:

$$SCI = (E \cdot I) + M \text{ per } R$$

Nella prima versione dell'adapter, avveniva solo il calcolo delle emissioni operative ($E \cdot I$). Per estendere il calcolo a tutto lo SCI sono state necessarie parecchie modifiche, tra cui:

- l'ottenimento delle embodied Emissions
- l'ottenimento della functional unit

In particolare, le operazioni necessarie per ottenere la functional unit variano a seconda della functional unit, così come possono essere calcolate functional unit diverse per applicazioni diverse.

Implementazione con Dynamic beans registration

Per implementare una functional unit e garantire che l'adapter sia facilmente estendibile a nuove functional unit, è stato necessario definire un meccanismo efficace, efficiente, che permetta di monitorare lo SCI abilitando la variabile d'ambiente `SCI_ENABLED`. Per raggiungere questo obiettivo è stato necessario implementare l'approccio di *Dynamic beans registration*:

> La Dynamic beans registration in Spring Boot si riferisce alla possibilità di registrare i bean a runtime, piuttosto che definirli staticamente nei file di configurazione o attraverso le annotations.

per farlo è stata definita una classe `FunctionalUnitSingletonRetriever`5, annotata con le annotazioni:

- `@Component` questa annotazione contrassegna la classe come componente gestito da Spring, il che significa che verrà automaticamente rilevata e registrata durante la scansione dei componenti. Spring creerà un'istanza singleton di questa classe e ne gestirà il ciclo di vita, consentendo di iniettarla in altri bean gestiti da Spring.
- `@ConditionalOnProperty(value = "adapter.sciEnabled", havingValue= "true")` specifica che il bean sarà creato solo se la variabile d'ambiente definita nell'`application.yml` chiamata adapter.sciEnabled sia a `true`.

il bean verrà creato solo se il valore della proprietà è `true`, quindi solo se lo SCI è abilitato. Questo permette di non sprecare memoria in caso di SCI disabilitato, in tal caso il bean non verrà creato.

Per quanto riguarda l'ottenimento della functional unit, è stato mantenuto lo stesso approccio dell'ottenimento delle metriche. Quindi è stata definita un'interfaccia `FunctionalUnitRetriever`

```
1 public interface FunctionalUnitRetriever {
2   /**
3    * Computes the functional unit for the given application result.
4    * @param applicationResult The application result for which the
     functional unit needs to be computed.
5    */
6   void computeFunctionalUnit(ApplicationResultDto applicationResult);
7 }
```

in generale il metodo `computeFunctionalUnit` opera tramite il riferimento di `applicationResult` per impostare il valore della functional unit tramite setter. La `FunctionalUnitSingletonRetriever` è un'implementazione del design pattern Singleton, nello specifico espone una `HashMap retrievers` con chiave una `FunctionalUnit` e come valore un oggetto che implementasse l'interfaccia `FunctionalUnitRetriever`. Il cuore della classe è il metodo `getRetriever`. 5

Se la functional unit in input non viene trovata nei `retrivers`, allora viene registrato un nuovo bean col `Retriever`, lo recupera dal contesto dell'applicazione Spring e lo aggiunge alla `HashMap`. Questo metodo assicura che i retriever per le diverse unità funzionali siano *lazily instantiated* e messi in cache, fornendo un modo efficiente per recuperarli in base alle functional unit, evitando al contempo istanziazioni non necessarie per la stessa functional unit.

Implementare una nuova functional unit

Ipotizziamo di aver definito una nuova functional unit `Foo`, dopo averla aggiunta all'enum

```
1 public enum FunctionalUnit {
2   Hourly500Requests,
3   HourlyLogin,
4   Foo
5 }
```

occorre creare una nuova classe `Retriever` per il recupero della functional unit `Foo`, i passi da seguire sono i seguenti:

1. Definire una classe `FooRetriever` che implementi l'interfaccia `FunctionalUnitRetriever`.
2. Implementare il metodo `computeFunctionalUnit(applicationResult)` all'interno di `FooRetriever`.
3. Aggiornare lo switch statement all'interno del metodo `getRetriever`5 di `FunctionalUnitSingletonRetriever` per gestire il caso in cui la functional unit sia `Foo`:
   (a) Registrate un nuovo bean di tipo `FooRetriever` usando `context.registerBean(FooRetriever.class, FooRetriever::new)`.
   (b) Aggiungere un riferimento al nuovo bean alla `HashMap` dei retrievers.

FunctionalUnitRetriever implementati fin'ora

Sono state definite due functional unit:

1. 500 chiamate API in un'ora, detta `Hourly500Requests`
2. Login in un'ora, detta `HourlyLogin`

per calcolarne il valore, sono state sviluppate due classi che implementano l'interfaccia `FunctionalUnitRetriever`:

1. `Hourly500RequestRetriever`
2. `HourlyLoginRetriever`

Osservazione. *Per evitare che questa parte risulti prolissa e ripetitiva verrà analizzata solo la* ***`Hourly500Requests`****.*

Per quanto riguarda la `Hourly500Requests`, sarà necessario recuperare il numero di richieste attraverso una query su una tabella di un database MySQL. La tabella in questione si chiama `API_CALL_HISTORY`, per interagire con essa è stata definita un'interfaccia `ApiCallHistoryRepository` che estenda l'interfaccia `Repository` nativa di Spring JPA:

```
@Lazy
@Repository
public interface ApiCallHistoryRepository extends Repository<
    ApiCallHistory, Long> {
    // code
}
```

- l'annotazione `@Lazy` indica che il bean creato da questa interfaccia deve essere inizializzato `lazy`. La Lazy initialization rinvia la creazione e l'inizializzazione di un bean fino a quando non viene richiesto per la prima volta.
- È parametrizzata con `<ApiCallHistory, Long>`, dove `ApiCallHistory` 5 è la classe di entità gestita da questo repository e `Long` è il tipo di chiave primaria di tale entità (uguale a quella definita nello schema del database).

all'interno di quest'interfaccia è stato definito un solo metodo:

```
@Query("""
    select
        sum(ach.requests_counter)
    from
        API_CALL_HISTORY ach
    where
        ach.execution_date = DATE(:execution_date)
""")
Double sumOfRequestsCounterOfExecutionDate(@Param("execution_date")
    Date execution_date);
```

- L'annotazione `@Query` proviene da Spring Data JPA ed è usata per definire query direttamente nell'interfaccia del repository.
- `@Param("execution_date")` è usato per specificare il parametro nominato nella query, che sarà sostituito con il valore passato al metodo.

questa query restituisce la somma dei valori della colonna `requests_counter` per i record della tabella `API_CALL_HISTORY` in cui la data di esecuzione corrisponde alla data di `:execution_date` fornita in input al metodo.

Questa repository viene utilizzata da `Hourly500RequestRetriever`:

```
1  public class Hourly500RequestRetriever implements
       FunctionalUnitRetriever {
2    private ApiCallHistoryRepository apiCallHistoryRepository;
3
4    @Autowired
5    public void setApiCallHistoryRepository(ApiCallHistoryRepository
       apiCallHistoryRepository) {
6      this.apiCallHistoryRepository = apiCallHistoryRepository;
7    }
```

- il metodo setter è annotato con `@Autowired`, indica che Spring dovrebbe iniettare un'istanza di `ApiCallHistoryRepository` in questa classe.
- quando Spring inizializza un oggetto di `Hourly500RequestRetriever`, cercherà un bean di tipo `ApiCallHistoryRepository` e lo inietterà in questo metodo setter.

Il calcolo della functional unit viene effettuato all'interno del metodo `computeFunctionalUnit`:

```
1  @Override
2  public void computeFunctionalUnit(ApplicationResultDto
       applicationResult) {
3    double defaultFunctionalUnitValue = 1d;
4    Double functionalUnitValue = apiCallHistoryRepository
5      .sumOfRequestsCounterOfExecutionDate(
6          applicationResult.getStartDate()
7      );
8    if (functionalUnitValue != null)
9      functionalUnitValue = functionalUnitValue / 12000;
10
11   addFunctionalUnitToResults(applicationResult,
12     Objects.requireNonNullElse(functionalUnitValue,
       defaultFunctionalUnitValue));
13 }
```

- utilizza l'`apiCallHistoryRepository` per ottenere il valore della functional unit specifico di tutta la giornata precedente.
- la funzione `addFunctionalUnitToResults` imposta il valore della functional unit su ogni `interval` all'interno di ogni servizio. Inoltre è stato definito un valore di default pari a `1.0`, nel caso non ci siano record nella tabella in tale data.

Il `FunctionalUnitSingletonRetriever` viene impiegato all'interno del metodo `run` della classe `BootStrapData`. Per ogni applicazione verrà eseguito questo codice:

```
1 if (adapterConfig.sciEnabled()) {
2     FunctionalUnit functionalUnit = application.getFunctionalUnit();
3     FunctionalUnitRetriever functionalUnitRetriever =
      functionalUnitSingletonRetriever.getRetriever(functionalUnit);
4     functionalUnitRetriever.computeFunctionalUnit(applicationResult);
5 }
```

l'implementazione di nuove `FunctionalUnit` non implicherà quindi la modifica di nessuna classe (tolti i passi necessari indicati in 3.4.5).

Osservazione. *Nella sezione 3.4.1 si è discusso dell'utilità di* ***`sci_enabled`*** *e* ***`sci_disabled`****. Se si è in* ***`application_disabled`****, la repository in 3.4.5 non verrà mai creata. Questo comportamente è dovuto all'annotazione* ***`@Lazy`****. Essendo lo scan dei bean di questo tipo disabilitato, nessuna astrazione di Spring considererà questo bean necessario per l'esecuzione dell'adapter.*

Embodied Emissions

Per calcolare le embodied emissions, si sfrutta l'endpoint `/v1/embodiedEmissions` del calcolatore, che richiede i seguenti parametri:

type il tipo di server

nCPUs numero di CPU sulla macchina

memorySize quantità di RAM

nSSD numero di SSD sulla macchina

nHDD numero di HDD sulla macchina

nGPU numero di GPU sulla macchina

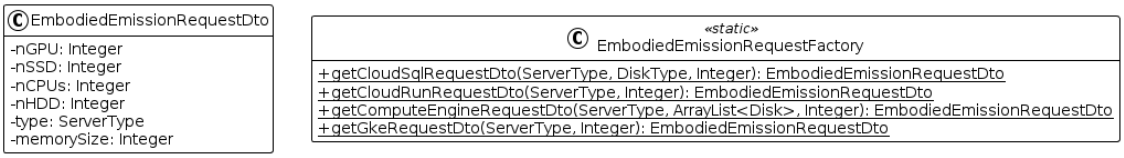
con questi dati è possibile calcolare la M di ogni layer. Le richieste al calcolatore cambiano in base al layer che si sta considerando:

Layer	Type	nCPUs	memorySize	nSSD*	nHDD*	nGPU
Cloud Run	`BLADE`	1	`memory`	`null`	`null`	`null`
Cloud SQL	`BLADE`	1	`ramSize`	1	1	`null`
Compute Engine	`BLADE`	1	max(`ramSize`)	`disks.size()`	`disks.size()`	`null`
GKE	`BLADE`	1	`memoryLimit`	`null`	`null`	`null`

*`nSSD` e `nHDD` sono mutuamente esclusivi, un layer non può utilizzare contemporaneamente dischi HDD ed SSD.

Sono state definiti una classe `EmbodiedEmissionRequestDto` che rappresenti la richiesta e una classe *static factory method* che faciliti la creazione di richieste:

la factory viene utilizzata all'interno di `BootStrapData` per creare la richiesta:




```
1 if (adapterConfig.sciEnabled()) {
2   EmbodiedEmissionRequestDto embodiedEmissionRequest =
     EmbodiedEmissionRequestFactory.getCloudRunRequestDto(
3           application.getType(), cloudRunResponse.getMemory());
4   EmbodiedEmissionResponseDto embodiedEmissionResponse = footprint.
     getEmbodiedEmission(embodiedEmissionRequest);
5   cloudRunResponse.setEmbodiedEmission(embodiedEmissionResponse);
6 }
```

che verrà poi inviata dal `FootprintService` al calcolatore, per recuperare l'embodied emission del layer.

3.4.6 Footprint Service

Il `FootprintService` è un'interfaccia che definisce un insieme di metodi per interagire con il calcolatore:

```
1 public interface FootprintService {
2     void postIngestionV2(ArrayList<ResponseDto> monitoringResponse);
3
4     ArrayList<Co2ResponseDto> postCo2(ArrayList<ResponseDto>
     monitoringResponse);
5
6     EmbodiedEmissionResponseDto getEmbodiedEmission(
     EmbodiedEmissionRequestDto embodiedEmissionRequest);
7
8     ArrayList<AggregatedCo2ResultDto> getEmissionAndPowerUsePerLayer(
     ArrayList<Co2ResponseDto> responseCo2);
9 }
```

è stata definita una sola implementazione di questa interfaccia, chiamata `FootprintServiceImpl`; oltre a implementare tutti i metodi definiti dall'interfaccia, definisce dei metodi privati che utilizza per la costruzione di richieste a partire da un array di risposte prodotto dai dati raccolti dai Retrievers.

Richieste http al calcolatore

Per inviare richieste http tramite java viene utilizzata la libreria `java.net.http`, in particolare:

- la richiesta viene creata attraverso l'interfaccia `HttpRequest`, costruita attraverso un `Builder`. È possibile impostare l'URI, le intestazioni e il corpo di una richiesta. I corpi delle richieste sono forniti attraverso un `BodyPublisher`

fornito a uno dei metodi POST, PUT. Tutte le richieste hanno questa struttura:

```
String requestBody = mapper.writeValueAsString(requestPojo);
HttpRequest request = HttpRequest.newBuilder()
  .uri(URI.create(endpoint))
  .header("x-api-key", adapterProperties.apikey())
  .header("x-token", adapterProperties.apikey())
  .header("Content-Type", "application/json")
  .POST(HttpRequest.BodyPublishers.ofString(requestBody))
  .build();
```

- l'interfaccia `HttpClient` per inviare richieste e recuperare le relative risposte. Viene creato attraverso un `builder`, con cui configurare il client. L'adapter usa il client di default instanziabile dal metodo `newHttpClient()`.

```
HttpResponse<String> response;
try (final HttpClient client = HttpClient.newHttpClient()) {
    // define request here ...
    response = client.send(request, HttpResponse.BodyHandlers.
    ofString());
}
```

le stringhe da inserire all'interno del body vengono generate attraverso dei `Static Request Factory`.

Static Request Factory

Sono state create tre *static factory*:

- `Co2RequestFactory` per creare il body delle richieste a `v1/carbonfootprint-/bulk/serverless/co2`
- `IngestionV2RequestFactory` per creare il body delle richieste a `v2/ingestion-event`
- `EmbodiedEmissionRequestFactory` per creare il body delle richieste a `v1/embodiedEmissions`

facilitano la creazione di richieste e non necessitano di essere istanziate.

Invio delle metriche al calcolatore

Il flusso d'esecuzione del metodo `run` della classe `BootStrapData` converge in questo *if-statement*:

```
if (adapterConfig.sendToCalculator()) {
  ArrayList<Co2ResponseDto> responseCo2 = footprint.postCo2(results);
  if (responseCo2 != null) {
      ArrayList<AggregatedCo2ResultDto> aggregatedResult =
```

```
5           footprint.getEmissionAndPowerUsePerLayer(responseCo2);
6     } else {
7         footprint.postIngestionV2(results);
8     }
9  }
```

Tutto dipende dalla variabile d'ambiente booleana `SEND_TO_CALCULATOR`:

- se `false`, le metriche contenute in `results` verranno inviate al database tramite ingestion. Quindi tutta la giornata verrà monitorata correttamente e apparirà in dashboard. Nello specifico il metodo `postIngestionV2` è il punto di ingresso principale per l'ingestione delle metriche di monitoraggio.
  - Esso invoca il metodo `sendAllIngestionV2`, passando l'elenco dei DTO di risposta al monitoraggio.
  - Itera su ogni risposta di monitoraggio, estrae le informazioni sui diversi layer (come Cloud Run, Cloud SQL, Compute Engine e GKE) e prepara i loro payload per l'ingestione.
  - Per ogni layer, costruisce i body delle richieste di ingestione usando i metodi della `IngestionV2RequestFactory` e invia le richieste post all'endpoint di ingestione tramite il metodo `sendPostRequestToIngestion`.
- se `true`, verrà contatto un servizio del calcolatore all'endpoint `v1/carbon-footprint/bulk/serverless/co2`. Il connettore invia i risultati al calcolatore, lui restituirà CO2 prodotta e consumo energetico di ogni singola ora per
  - Storage
  - CPU
  - GPU
  - Memory

  di ogni layer. Attraverso il metodo `getEmissionAndPowerUsePer-Layer` viene eseguita una *group by layer* che sommi la CO2 emessa e il consumo energetico giornaliero per ogni layer. I risultati verranno stampati su stdout del Cloud Run.

Questa capacità di calcolare le emissioni e i consumi senza dover ingestire dati è stata fondamentale durante la fase di collaudo del connettore. Ha permesso di testare il connettore senza alterare il database con ingestioni eseguite localmente, che avrebbero sporcato il monitoraggio giornaliero.

3.5 Utilizzo della pipeline CI/CD

Durante il ciclo di vita del progetto, il team ha fatto uso di pratiche DevOps che consentono la rapida e affidabile consegna di modifiche al codice. In particolare, è stata sviluppata una pipeline CI/CD descritta nell'immagine seguente.

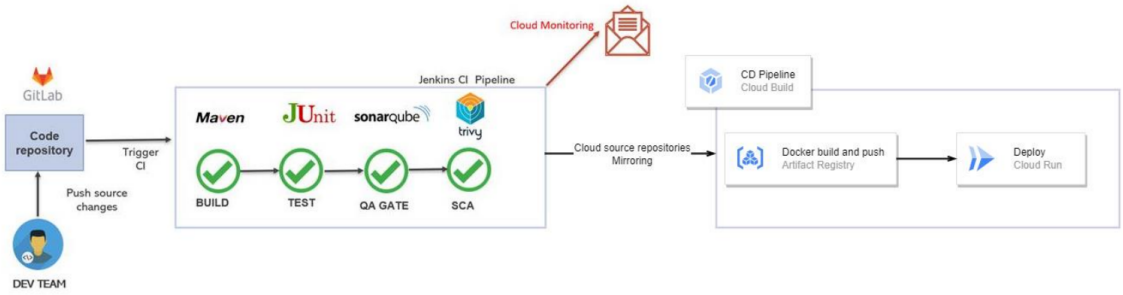


Figura 3.7: Pipeline CI/CD DevOps con Repository, Build Jenkins, Test, Quality Gate, Analisi Software Composition, Build Cloud, Artifact Registry e Deploy Cloud

Ad ogni push sul codice sorgente, viene eseguita una pipeline CI di GitLab. Questa pipeline automatizza i seguenti task:

Build utilizzando Maven.

Esecuzione dei test unitari utilizzando JUnit.

Quality Gate utilizzando SonarQube. Il codice viene analizzato per verificare che la copertura superi l'80%, che la qualità del codice sia adeguata (valutazione dell'affidabilità, manutenibilità e conteggio dei blocchi di codice duplicati) e che non vengano rilevati problemi di sicurezza mediante il SAST (ad esempio vulnerabilità di sicurezza, hotspot di sicurezza, che sono punti chiave sensibili alla sicurezza che richiedono una revisione manuale per valutare se esista o meno una vulnerabilità). Il test di sicurezza delle applicazioni statiche (SAST) è un insieme di tecnologie progettate per analizzare il codice sorgente dell'applicazione, il bytecode e i binari per condizioni di codifica e di progettazione che sono indicative di vulnerabilità di sicurezza. Le soluzioni SAST analizzano un'applicazione dall'*interno verso l'esterno* in uno stato non in esecuzione.

SCA utilizzando Trivy. Il codice viene analizzato per valutare la sicurezza, la conformità alle licenze e la qualità del codice delle librerie utilizzate. Trivy è uno scanner di sicurezza completo e versatile. Trivy dispone di scanner che ricercano problemi di sicurezza e obiettivi in cui è possibile trovare tali problemi. Inoltre, Trivy può controllare immagini di container, filesystem, repository Git, immagini di macchine virtuali e diverse altre come pacchetti OS e dipendenze software in uso (SBOM).

Capitolo 4

Risultati

Il connettore viene utilizzato per monitorare le emissioni di CO2 e i consumi dell'intero ecosistema Green IT. In pratica monitora le emissioni di ogni singolo componente di Green IT: calcolatore, dashboard (frontend e backend), database, cluster kubernetes, etc.

4.1 Utilizzo della dashboard

Green IT mette a disposizione una dashboard suddivisa in più sezioni. Ci sono quattro filtri che possono modificare le informazioni mostrate nei widget:

- una finestra temporale
- `Application`, un insieme di applicazioni
- `Layer`, un insieme di componenti delle applicazioni selezionate
- `Functional units`, permette di filtrare solo le functional unit dei layer selezionat

Questi parametri fungeranno da filtro, verranno mostrati a schermo solo dati che soddisfano questi vincoli.

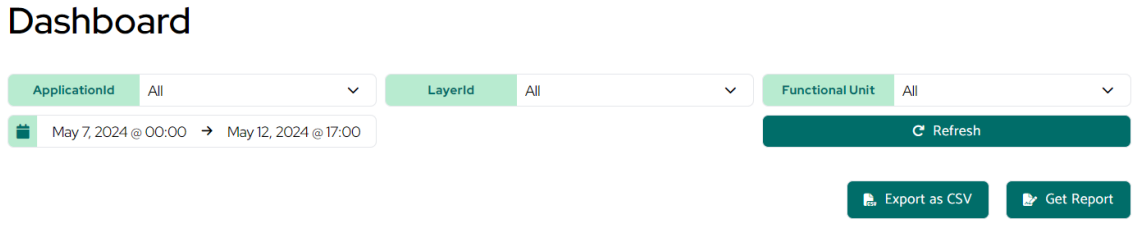


Figura 4.1: Fase di selezione

Widgets

Quattro quadranti

Questo widget è visibile solo se lo SCI è abilitato. Il widget dei quattro quadranti fornisce una panoramica visiva del rapporto tra le emissioni di CO2 e lo SCI di ogni applicazione. Presenta un grafico diviso in quattro quadranti da due rette con all'interno dei punti. Le rette rappresentano la media per applicazione dello CO2 e dello SCI, ogni punto invece rappresenta la singola applicazione.

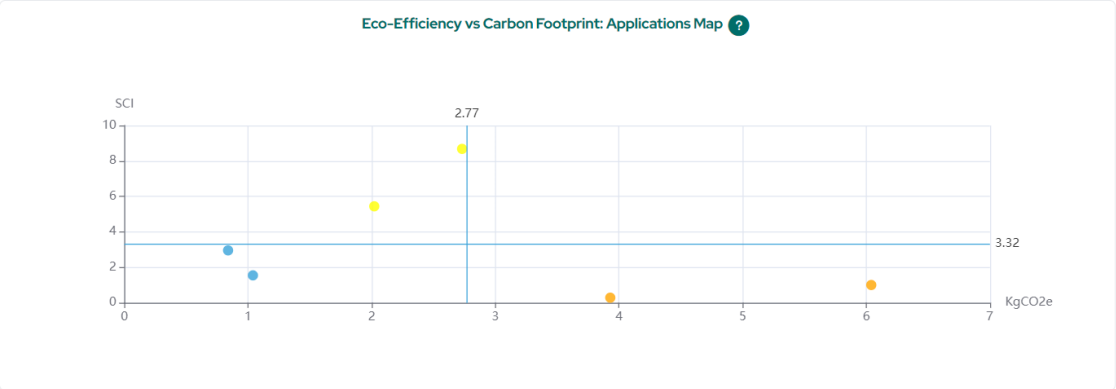


Figura 4.2: Widget dei quattro quadranti

Quadrante in basso a sinistra contiene gli *elementi tranquilli*, dove i valori di CO2 e SCI sono bassi.

Quadranti in alto a sinistra/basso a destra rappresentano *elementi semi-critici*, in quanto presentano valori elevati di uno dei due parametri.

Quadrante in alto a destra sono considerati *elementi critici*, in quanto presentano valori elevati sia di CO2 che di SCI.

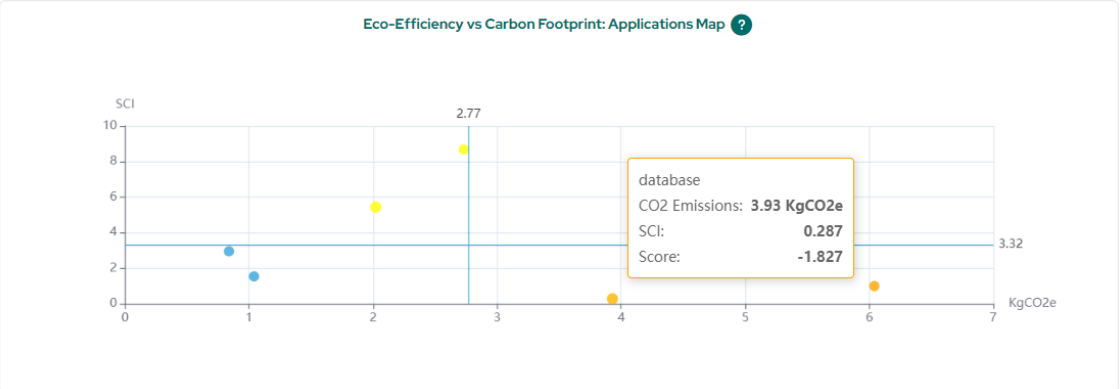


Figura 4.3: Dettaglio sul valore dell'applicazione Database

Nel caso di elementi semi-critici, possiamo quindi approfondire le due situazioni:

Alta CO2, basso SCI si parla degli elementi nel quadrante in basso a destra, sono applicazioni molto utilizzate ma efficienti in termini di emissioni prodotte per operazione. Nell'immagine fornita sopra, l'applicazione con il valore più alto di CO2 in questo quadrante ha uno SCI inferiore alla media.

Bassa CO2, alto SCI si parla degli elementi nel quadrante in alto a sinistra, sono applicazioni poco sollecitate, quindi con basse emissioni globali, ma con elevato consumo di CO2 per operazione.

Per quanto riguarda gli **elementi critici** abbiamo **alto CO2 e alto SCI**, quindi applicazioni molto utilizzate e con elevato impatto ambientale per operazione, che necessitano quindi di essere ottimizzate.

Widget a serie temporali per lo SCI totale

Questo grafico mostra:

- con una linea blu il variare del valore dello SCI durante la finestra di tempo selezionata
- con una linea gialla le embodied emissions della macchina per la finestra di tempo selezionata

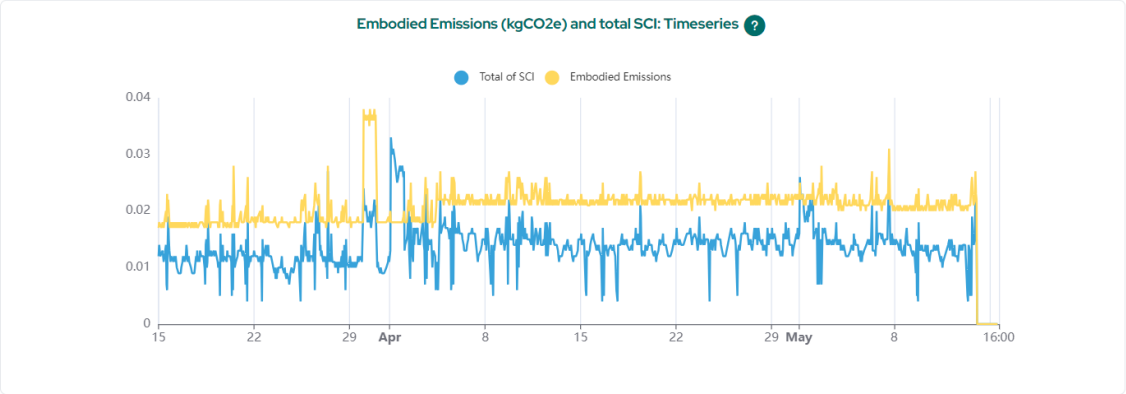


Figura 4.4: Timeseries che mostra il totale dello SCI

Questo widget visualizza le emissioni di CO2 del server in due modi: le emissioni dei suoi singoli componenti (CPU, GPU, memoria e storage) e le emissioni totali del server rappresentate dalla linea blu, vista come la somma di tutte le emissioni dei componenti.

Piechart con la CO2 emessa per risorsa

Il widget a sinistra mostra le peggiori 10 applicazioni ordinate per valore di SCI. Invece il widget a destra presenta un grafico a torta che mostra la distribuzione percentuale delle emissioni di CO2 tra le risorse: CPU, GPU, STORAGE e MEMORIA. Offre una visione del contributo di ciascun tipo di risorsa alle emissioni totali di CO2, consentendo agli utenti di identificare le aree in cui è possibile ottimizzare e migliorare l'efficienza.

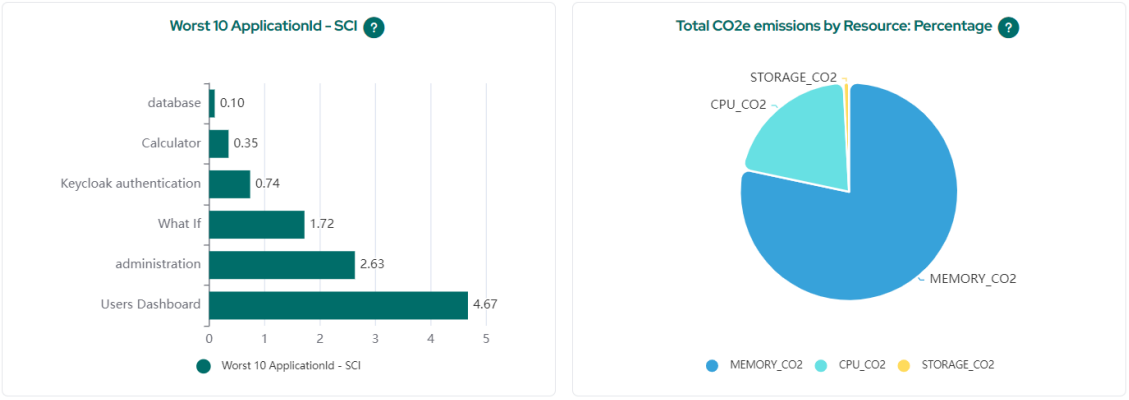


Figura 4.5: A sinistra le peggiori applicazioni per SCI, a destra un piechart che mostra la distribuzione percentuale delle emissioni

Timeframe per CO2 e SCI

Figura 4.6: Timeframe per le emissioni totali di CO2 e lo SCI medio

Il contatore a sinistra mostra la quantità totale di chilogrammi di CO2 emessi nella finestra di tempo selezionato da tutte le applicazioni selezionate. Il contatore a destra è visibile solo se vi sono informazioni riguardo lo SCI, mostra il valor medio dello SCI nella finestra di tempo selezionata.

Widget a serie temporali per la CO2 emessa per tipo

Questo widget fornisce una rappresentazione delle serie temporali delle emissioni di CO2 per le risorse `CPU`, `GPU`, `STORAGE` e `MEMORY`. Consente agli utenti di seguire le tendenze storiche delle emissioni di CO2 associate a ciascun tipo di risorsa nel tempo, permettendo di identificare i periodi in cui una particolare risorsa ha emesso più CO2.

Widget a barre

Il widget a sinistra mostra i 10 layer di un'applicazione che producono più CO2, in pratica i 10 peggiori. Ogni barra rappresenta un layer, e se è stata selezionata un'ap-

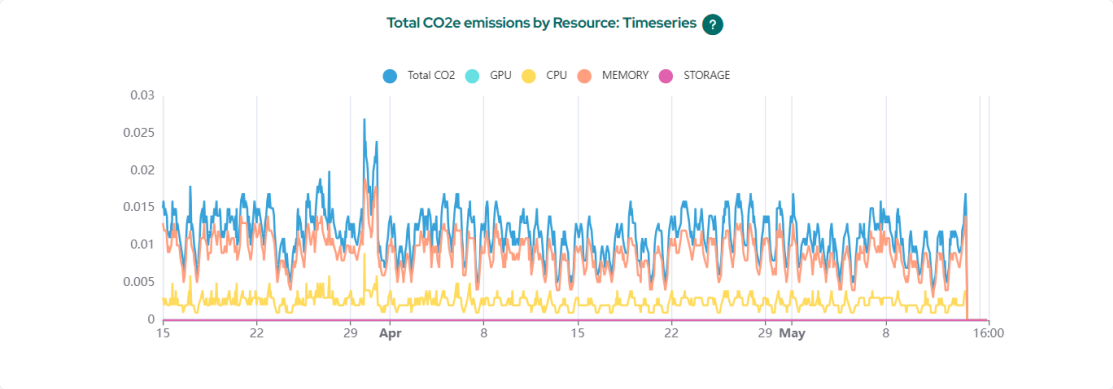


Figura 4.7: Timeseries che mostra le emissioni di CO2 per tipo di risorsa

plicazione specifica verranno mostrati solo i layer di quell'applicazione, altrimenti verranno mostrati tutti i layer. Il widget a destra Il widget presenta un istogramma che illustra la *carbon intensity* media nelle regioni in cui viene eseguito il software (rappresentato in tonalità più scure).

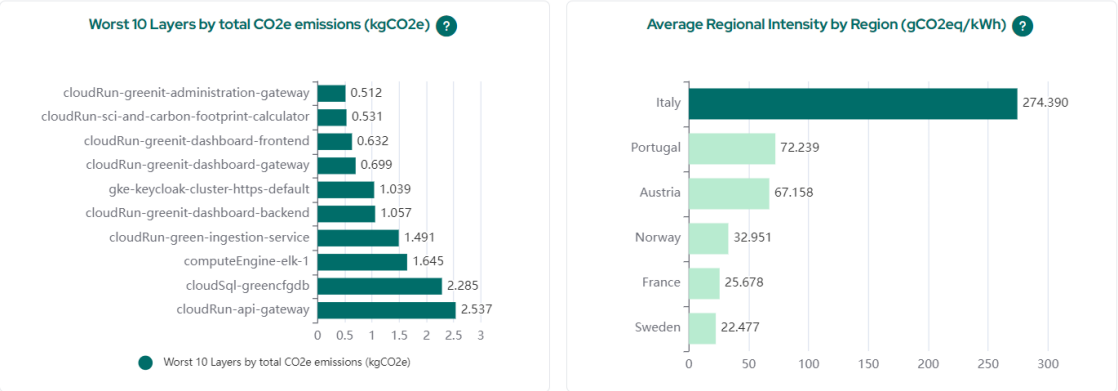


Figura 4.8: A sinistra i peggiori 10 layer per CO2 prodotta, a destra la carbon intensity media per regione in cui si trovano le macchine di google che vengono utilizzate per Green IT

4.2 Analisi sui risultati ottenuti

La seguente sezione è dedicata al confronto dei dati raccolti dal connettore nei mesi di Marzo e Aprile 2024. Prima di passare all'analisi dei dati, è bene chiarire alcune cose del periodo preso in analisi.

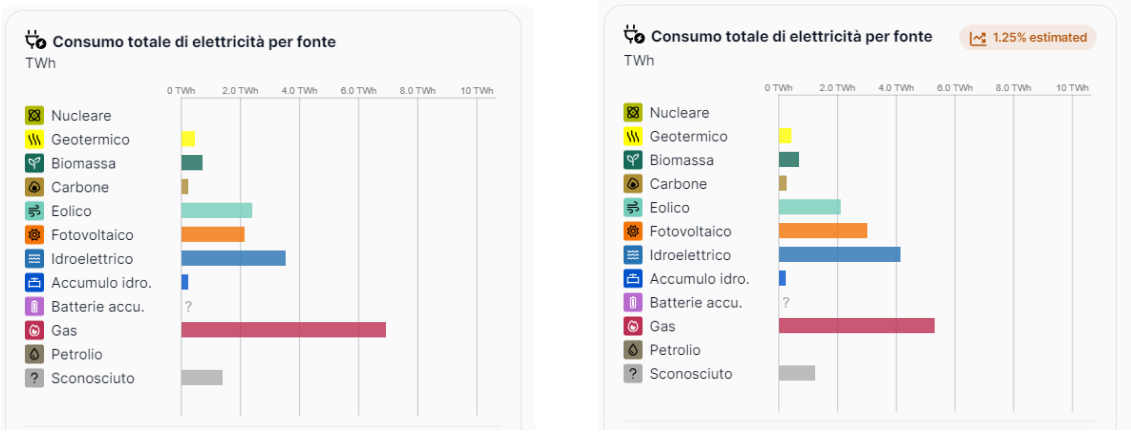
Marzo e Aprile

In tutta Italia sia Marzo che Aprile sono stati mesi con poche giornate di sole, ciò ha portato a una riduzione della produzione di energia solare che ha implicato il dover fare affidamento ad altre fonti energetiche.

Tabella 4.1: Dati Meteo di Milano [16]

	T Media	T min	T max	Umid.	Vento Med.	Giorni Pioggia
Marzo	11.5 °C	7.6 °C	15.3 °C	77.8 %	8.6 km/h	18
Aprile	14.4 °C	9.3 °C	19.5 °C	70.4 %	8.2 km/h	11

Come fonti green in Italia domina l'idroelettrico da sempre, seguono il fotovoltaico, le bioenergie, l'eolico e il geotermico [18]. Utilizzando la piattaforma online Electricity Maps è possibile visualizzare dati su produzione e consumi di elettricità in diverse regioni del mondo, inoltre mostra la provenienza dell'energia elettrica (fonti rinnovabili, combustibili fossili, nucleare) e carbon intensity [17]. Utilizzando questo



(a) Marzo (b) Aprile

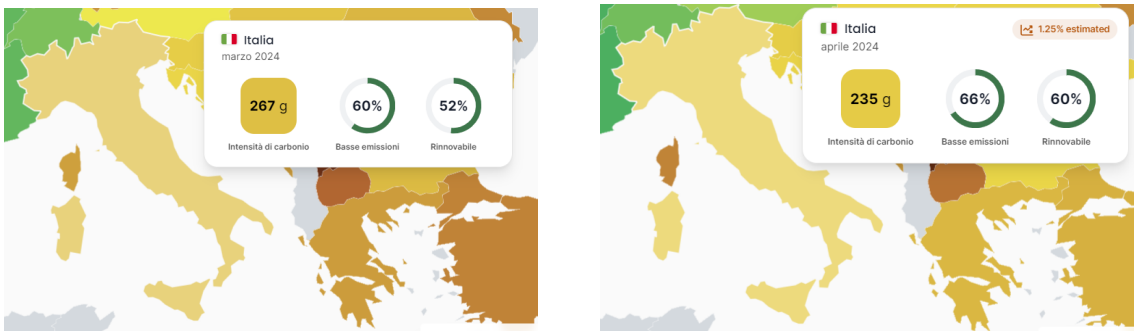
Figura 4.9: Consumo di elettricità in Italia per fonte, in (a) meno rinnovabili di (b)

servizio è possibile notare come a Marzo l'Italia ha consumato molta più energia da fonti non rinnovabili rispetto che ad Aprile:

- è stato usato molto più gas (non rinnovabile)
- è stato usato meno idroelettrico (rinnovabile)
- è stato usato meno fotovoltaico (rinnovabile)

- è stato usato più eolico (rinnovabile)

Questo si traduce in un aumento della Carbon Intensity a Marzo, perché si usa meno energia rinnovabile (fotovoltaico, eolico, idroelettrico) e più combustibili fossili (gas naturale), rilasciando una quantità molto maggiore di CO2 nell'atmosfera rispetto alle fonti rinnovabili.



(a) Marzo (b) Aprile

Figura 4.10: Carbon Intensity ed energia rinnovabile consumata in Italia

L'aumento della Carbon Intensity ha un impatto diretto sullo SCI, visto che nella formula le emissioni operative O dipendono da energia consumata E e Carbon Intensity I. Queste differenze verranno osservate meglio nelle analisi.

Emissioni

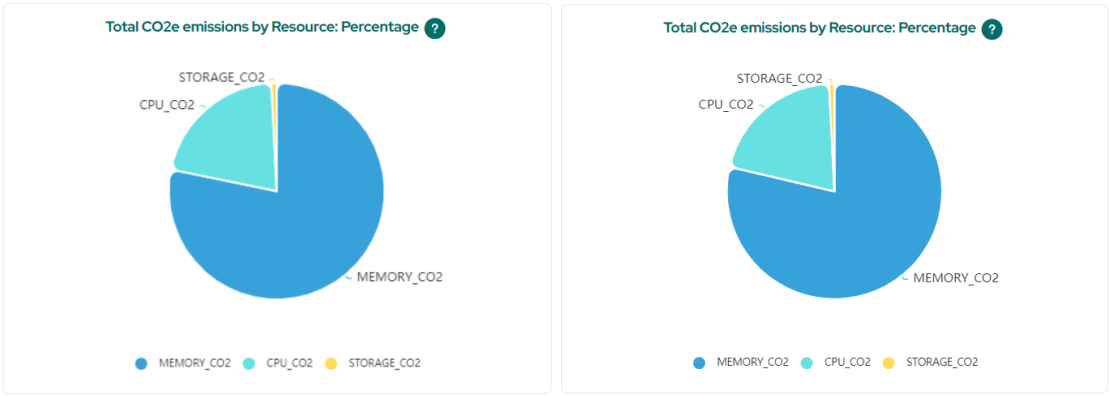
In entrambi i mesi l'anidride carbonica emessa è molto bassa, questo perché tutti i componenti utilizzati da Green IT sono molto efficienti e *green*. Bisogna tenere conto anche che Aprile ha un giorno in meno rispetto a Marzo. In generale 9.6

(a) Marzo (b) Aprile

Figura 4.11: Tachimetro della CO2e totale emessa, (a) ha prodotto di più di (b)

kgCO2e emessi in un mese è un valore relativamente basso per un software.

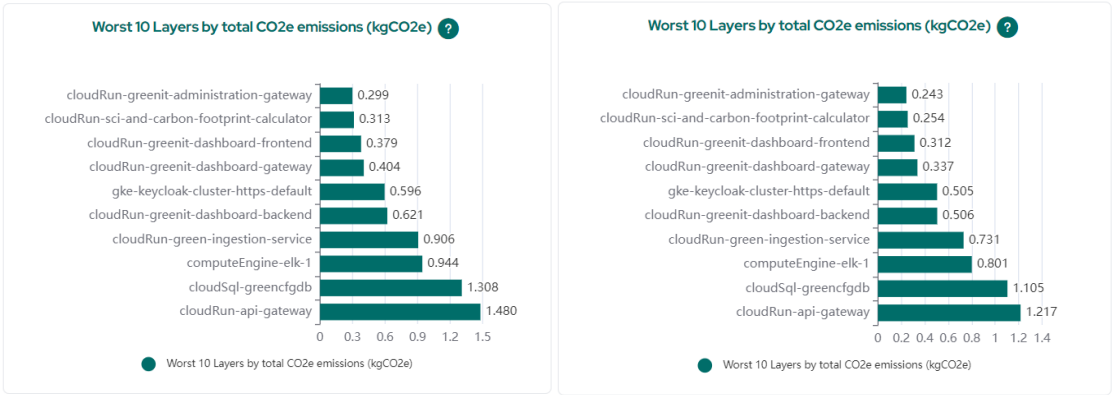
Dai piechart sottostanti si evince che:



(a) Marzo (b) Aprile

Figura 4.12: Distribuzione percentuale della CO2e emessa per tipo di risorsa

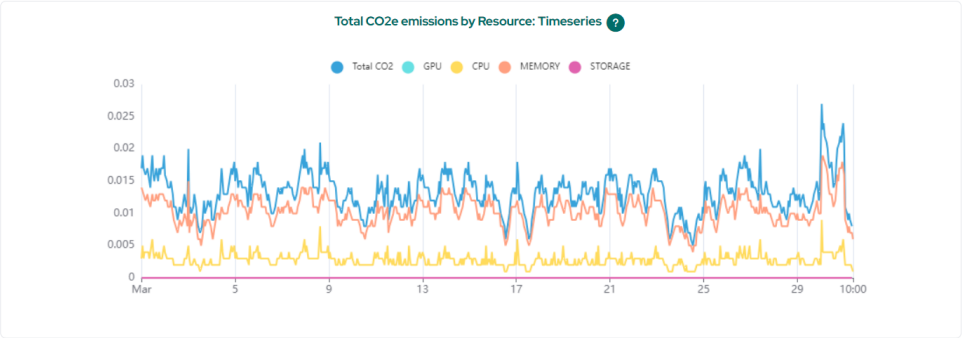
- la risorsa che ha emesso più CO2e in assoluto è la memoria, questo perché l'istanza Cloud SQL utilizzata da Green IT viene sollecitata in molti momenti della giornata e utilizza molta RAM. Le RAM sono dispositivi di memoria volatile, significa che richiedono un flusso costante di energia per mantenere i dati.
- a seguire abbiamo la CPU, che emette abbastanza CO2e soprattutto a causa delle numerose istanze di Compute Engine e Cloud Run che utilizza Green IT.
- la risorsa che emette meno è lo STORAGE, perché gli Hard Disk consumano poco (rispetto alle RAM ad esempio) e consumano energia solo quando leggono o scrivono dati.
- nessun componente di Green IT utilizza GPU.
- entrambi presentano pressocché le medesime percentuali, quindi possiamo affermare che la CO2 emessa rispetto al tipo di risorsa è rimasta invariata.



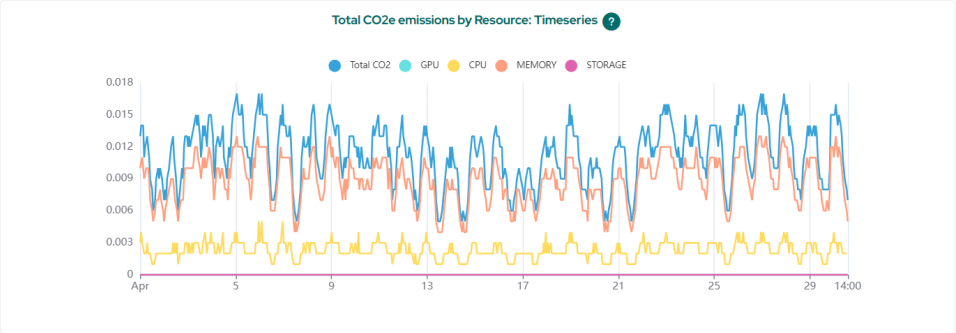
(a) Marzo (b) Aprile

Figura 4.13: I 10 layer che hanno causato più emissioni di CO2

Dal confronto fra le due timeseries, è palese che Marzo è stato un mese soggetto a una Carbon Intensity maggiore rispetto ad Aprile.



(a) Marzo



(b) Aprile

Figura 4.14: Timeseries con la CO2e totale emessa per tipo di risorsa

L'elettricità consumata ad Aprile è più pulita di quella consumata a Marzo, sicuramente quest'ultimo ha avuto meno giornate di sole quindi meno fotovoltaico a disposizione. L'andamento del grafico è abbastanza simile in entrambi, ma quello di Marzo è più allungato sulle ordinate perché presenta dei picchi più ampi.

Il confronto 4.13 è un'ulteriore prova che ad Aprile Green IT ha prodotto meno emissioni rispetto a Marzo, ma nonostante ciò le posizioni della classifica sono identiche in entrambi i mesi. A questo punto è necessario specificare perché alcuni layer consumano più di altri, mi soffermerò solo sui primi tre:

cloudRun-api-gateway in generale un API Gateway è un componente che si colloca tra client e servizi backend. Ha diversi compiti: routing delle richieste, autenticazione e autorizzazione, caching... Di conseguenza è uno dei componenti più attivi di Green IT, visto che deve gestire centinaia di richieste al minuto.

cloudSql-greenfgdb è un database, i consumi sono legati all'hardware sottostante soprattutto la RAM.

computeEngine-elk-1 è una macchina virtuale su cui è installato lo stack ELK[1], viene utilizzato per centralizzare e gestire i log provenienti dall'intera

[1]Elastic Logstash Kibana

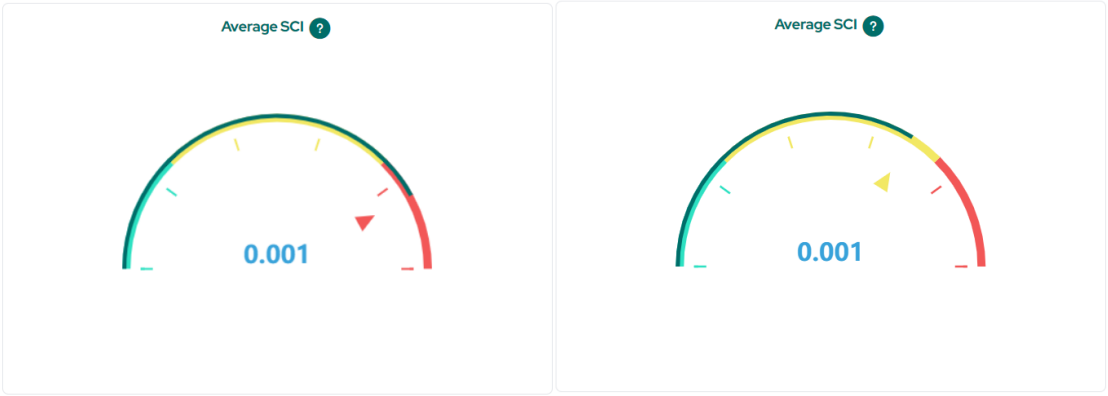
infrastruttura Green IT. E' uno stack molto potente ed efficiente, le emissioni sono dovute al fatto che la macchina virtuale è sempre accesa per garantire che nessun log venga perso.

SCI

Lo SCI si calcola attraverso la formula

$$(E \cdot I) + M \text{ per } R$$

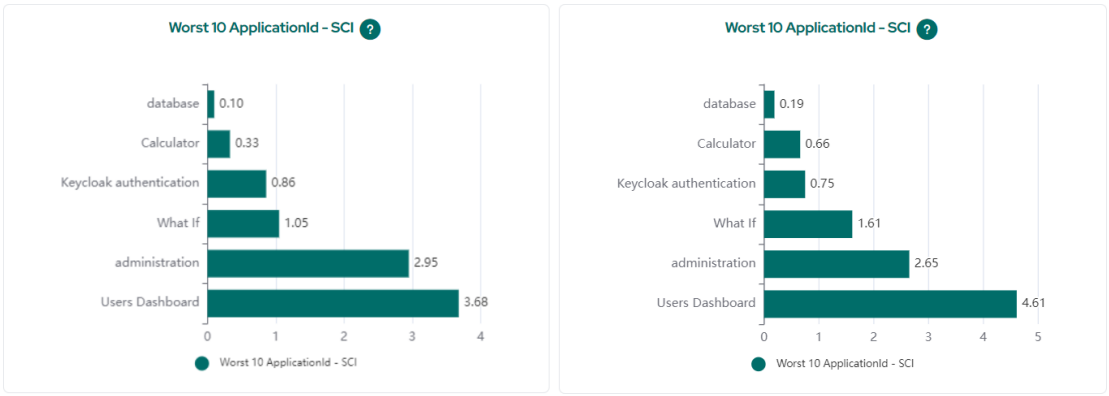
in entrambi i mesi abbiamo lo stesso ottimo valore medio di SCI.



(a) Marzo (b) Aprile

Figura 4.15: Valore medio dello SCI, uguali ma il massimo di (a) è minore di quello di (b)

Un valore medio di 0.001 indica che in media ogni componente di Green IT genera 0.001 kgCO2e rispetto alle Functional Unit di riferimento, è un valore molto basso e suggerisce che Green IT ha un impatto ambientale molto ridotto.

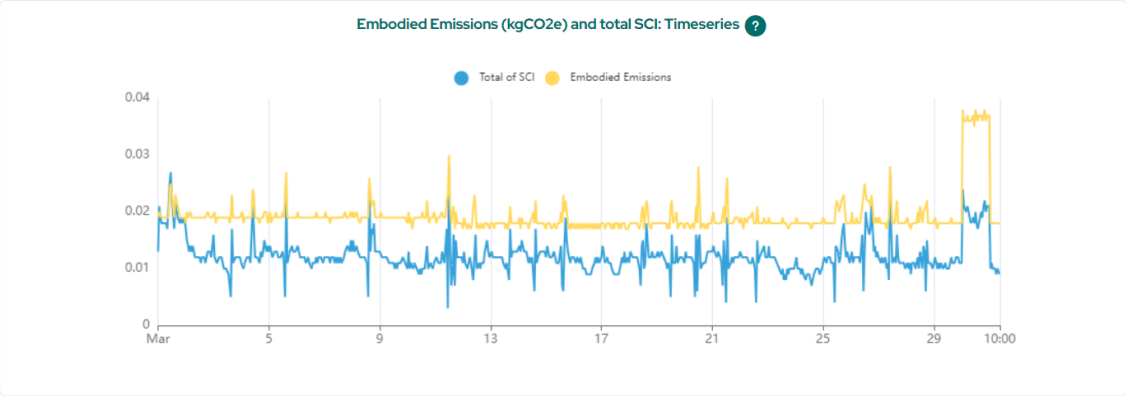


(a) Marzo (b) Aprile

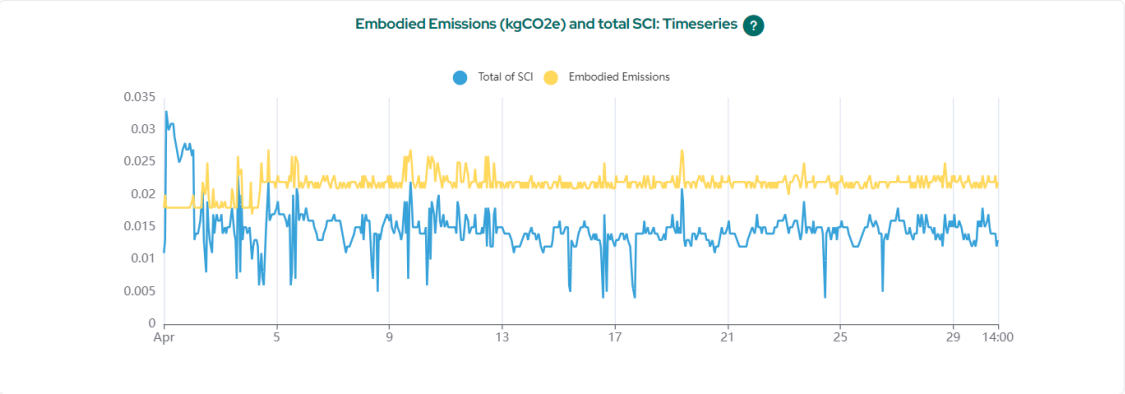
Figura 4.16: I 6 layer che hanno valori di SCI più alti

Embodied emissions e SCI hanno andamenti simili, perché entrambi sono legati al carico del servizio. Tuttavia l'embodied emissions sono legate all'hardware, quindi

aumentano quando un componente di Green IT necessiterà di fare *scale up* (maggiore provisioning di risorse). Invece lo SCI è sì legato allo scaling ma attraverso la Functional Unit R, questo spiega come mai l'andamento tra embodied emissions e SCI sia uguale ma scalato di fattori diversi.



(a) Marzo



(b) Aprile

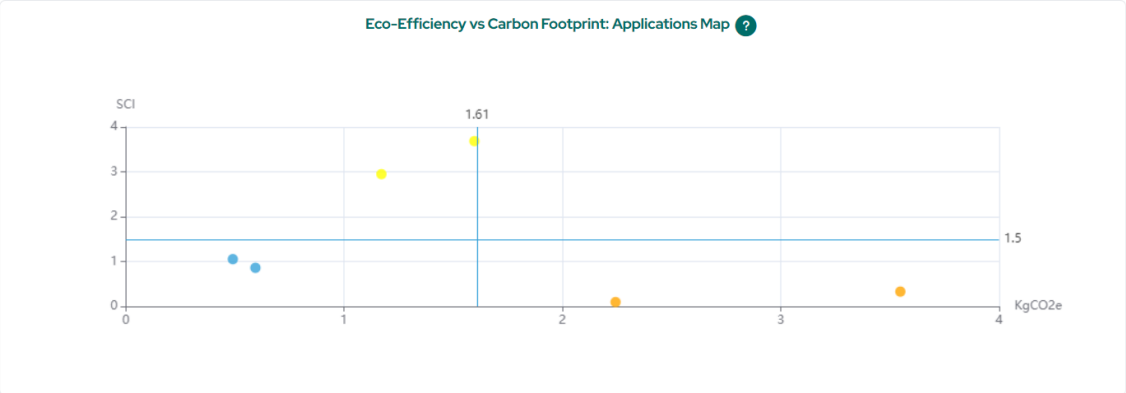
Figura 4.17: Timeseries con Embodied emission (giallo) e SCI totale (blu)

Riguardo al comportamento avuto a Marzo, sono presenti dei picchi abbastanza evidenti sull'embodied emissions. Questa cosa non si riflette su Aprile, infatti è visibilmente più stabile; questo è dovuto a un aumento delle risorse allocate da parte del team di sviluppo, questo comporta un carico molto più stabile (embodied emissions lineari) a fronte di uno SCI rimasto invariato (a parte qualche oscillazione dovuta alla Carbon Intensity).

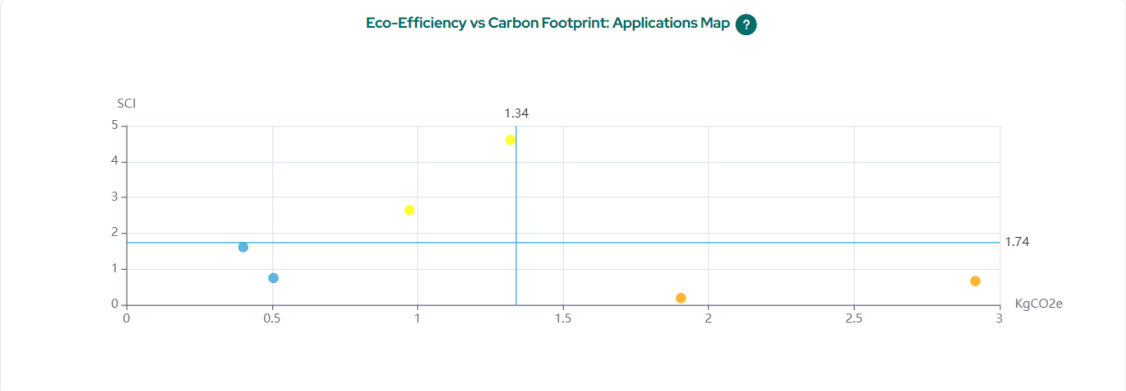
Dal grafico dei quattro quadranti si evince che:

- le applicazioni nel quadrante in basso a sinistra vengono dette *tranquille* hanna basso SCI e basso CO2 (miglior caso). La prima applicazione, chiamata What If, è un servizio di previsione delle emissioni che offre Green IT mentre la seconda è un cluster GKE con cui, attraverso Keycloak[2], viene gestita l'autenticazione di Green IT . Queste due applicazioni rappresentano il miglior compromesso in assoluto, hanno una buona efficienza e consumi molto bassi.

[2]Piattaforma software che permette la gestione dell'identità e degli accessi



(a) Marzo



(b) Aprile

Figura 4.18: Widget dei quattro quadranti, divisione simile ma (a) ha un valore di SCI più basso e di CO2 più alto di (b)

- le applicazione nei quadranti in alto a sinistra/in basso a destra vengono dette *semi-critiche*.
  - Le applicazioni nei quadranti in alto a sinistra hanno uno SCI molto alto quindi costo per operazione importante ma hanno una bassa efficienza. Da notare infatti che in entrambi i mesi, l'applicazione più in alto tende verso il quadrante in alto a destra.
  - Le applicazioni nei quadranti in basso a destra sono molto lontane dal quadrante della criticità, perché hanno consumi importanti ma sono molto efficienti.
  - non sono presenti applicazioni *critiche*, questo vuol dire che non ci sono applicazioni su cui dover eseguire dei lavori per migliorarne l'efficienza. Magari sarebbe da considerare migliorabile l'applicazione *borderline* con SCI più alto.

L'unica differenza di questo grafico tra i due mesi sono i valori medi di CO2 e di SCI. Aprile ha minor valor medio di CO2 (1.34) e maggior valor medio di SCI (1.76), questo indica che si comporta molto meglio di Marzo dal punto di vista delle

emissioni e dell'efficienza. Questo è dovuto alla Carbon Intensity minore del mese di Aprile, probabilmente perché ci sono stai più giorni di sole.

4.3 Considerazioni finali

L'analisi dei dati raccolti dal connettore Google Cloud nel periodo Marzo-Aprile 2024 ha evidenziato un impatto ambientale complessivamente ridotto per Green IT, grazie all'utilizzo di componenti software efficienti e a basso consumo energetico. Tuttavia, le differenze nelle condizioni meteorologiche tra i due mesi hanno influenzato la carbon intensity dell'energia elettrica utilizzata, con un conseguente aumento delle emissioni di CO2 a Marzo rispetto ad Aprile. Questo dimostra l'importanza di considerare la variabilità delle fonti energetiche nel calcolo dello SCI e nella valutazione dell'impatto ambientale del software. Nonostante le variazioni nelle emissioni, l'analisi dei layer e dei quadranti SCI ha evidenziato un comportamento stabile dell'infrastruttura software, senza la presenza di applicazioni critiche dal punto di vista dell'efficienza energetica.

Capitolo 5

Conclusioni

Questo tirocinio è stato completo sotto ogni punto di vista, permettendomi di approfondire non solo concetti legati al cloud e alla programmazione Java orientata agli oggetti, ma anche di acquisire una maggiore consapevolezza riguardo alla tecnologia sostenibile e alle sue implicazioni nel settore IT. In particolare, l'esperienza mi ha sensibilizzato sui rischi e le conseguenze dell'inquinamento prodotto dall'ICT, rendendomi consapevole dell'importanza di sviluppare soluzioni software che minimizzino l'impatto ambientale.

Sono estremamente soddisfatto del lavoro svolto, in quanto il connettore da me realizzato è tutt'ora utilizzato in azienda, fornendo quotidianamente dati che vengono sì mostrati durante workshop aziendali e presentazioni importanti, ma soprattutto sono serviti per capire quali punti dell'ecosistema Green IT andare a colpire per migliorarne la sostenibilità.

Lavorare su qualcosa da zero è stata una bella sfida, soprattutto implementare lo SCI. Il team di sviluppo mi ha lasciato *carta bianca*, permettendomi di studiare e implementare la miglior soluzione possibile che garantisse codice di qualità per gli sviluppi futuri.

I prossimi sviluppi di questo connettore riguarderanno l'introduzione di nuove Functional Unit per lo SCI e un refactoring di alcune classi *grezze* della codebase, così da migliorare la leggibilità, la manutenibilità e l'estensibilità del codice.
Questa realizzazione concreta mi ha permesso di mettere in pratica le conoscenze acquisite durante la triennale e di acquisirne nuove molto più verticali su specifici ambiti. Concludo il tirocinio con un bagaglio di competenze tecniche e trasversali notevolmente arricchito, una maggiore consapevolezza delle mie capacità e una visione più concreta del mondo del lavoro.

Bibliografia

[1] United Nations. Climate Change 2023. [Online]. Available: https://www.unep.org/resources/report/climate-change-2023-synthesis-report

[2] Unione Europea. Accordo di Parigi sui cambiamenti climatici. [Online]: Available https://www.consilium.europa.eu/it/policies/climate-change/paris-agreement/

[3] Life Gate. Cos'è la CO2 e come ridurla. [Online]. Available: https://www.lifegate.it/co2-definizione-come-ridurla

[4] GreenBiz. Microsoft tackles rising Scope 3 emissions. [Online]. Available https://www.greenbiz.com/article/microsoft-launches-initiative-counter-30-rise-scope-3-emissions-2020

[5] ONU. Emissions Gap Report 2023. [Online]. Available: https://img.esg360.it/wp-content/uploads/2023/11/21155317/EGR2023.pdf

[6] Apple. Environmental Progress Report 2022. [Online]. Available: https://www.apple.com/environment/pdf/Apple_Environmental_Progress_Report_2022.pdf

[7] Net 0 Tracker. APPLE INC. tracking [Online]. Available: https://net0tracker.org/corporates.html/Apple%20Inc./

[8] Google. What is IaaS? [Online]. Available: cloud.google.com/learn/what-is-iaas

[9] Google. What is Platform-as-a-Service? [Online]. Available: cloud.google.com/learn/what-is-paas

[10] Red Hat. What is CaaS? [Online]. Available: https://www.redhat.com/en/topics/cloud-computing/what-is-caas

[11] Google. Monitoring Query Language overview. [Online]. Available: https://cloud.google.com/monitoring/mql

[12] Spring. Dependency Injection. [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html

[13] Google. instance/cpu/usage_time API specification . [Online] Available: https://cloud.google.com/monitoring/api/metrics_gcp

[14] Spring. Profiles. [Online]. Available: https://docs.spring.io/spring-boot/reference/features/profiles.html

[15] Jackson. What is Jackson?. [Online]. Available: https://github.com/FasterXML/jackson?tab=readme-ov-file#what-is-jackson

[16] IlMeteo. Meteo di Milano di Marzo e Aprile 2024. [Online]. Available: https://www.ilmeteo.it/portale/archivio-meteo/Milano/2024

[17] Electricity Map. Applicazione. [Online]. Available: https://app.electricitymaps.com/zone/IT

[18] Enel. Le energie rinnovabili in Italia. [Online]. Available: https://www.enelgreenpower.com/it/learning-hub/energie-rinnovabili

Appendice codici sorgente

[5] application.yml

```yaml
adapter:
  bucketName: ${BUCKET_NAME}
  configFileName: ${CONFIG_FILE_NAME:config.json}
  apikey: ${APIKEY}
  sendToCalculator: ${SEND_TO_CALCULATOR:false}
```

[6] FunctionalUnitSingletonRetriever

```java
/**
 * Singleton class responsible for retrieving functional unit
   retrievers based on the specified functional unit.
 * This class manages the instantiation and retrieval of retrievers for
    various functional units.
 */
@Slf4j
@Component
@ConditionalOnProperty(
    value = "adapter.sciEnabled",
    havingValue = "true"
)
public class FunctionalUnitSingletonRetriever {
  private final Map < FunctionalUnit,
  FunctionalUnitRetriever > retrievers = new HashMap < >();

  private final GenericApplicationContext context;

  @Autowired
  public FunctionalUnitSingletonRetriever(GenericApplicationContext
    context) {
    this.context = context;
  }

  /**
```

```
23        * Retrieves the appropriate functional unit retriever based on the
          specified functional unit.
24        * If the retriever is not found in the cache, it instantiates and
          registers a new one.
25        * @param functionalUnit The functional unit for which the
          retriever is requested
26        * @return The retriever instance corresponding to the functional
          unit
27        * @throws IllegalStateException If an unexpected functional unit
          is provided
28        */
29      public FunctionalUnitRetriever getRetriever(FunctionalUnit
          functionalUnit) {
30       FunctionalUnitRetriever retriever = retrievers.get(functionalUnit);
31
32       if (retriever == null) {
33         log.info("{} not found in Map<FunctionalUnit,
          FunctionalUnitRetriever> retrievers", functionalUnit);
34         switch (functionalUnit) {
35         case HourlyLogin - >{
36             log.info("Instantiating HourlyLoginRetriever");
37             context.registerBean(HourlyLoginRetriever.class,
          HourlyLoginRetriever::new);
38             retriever = context.getBean(HourlyLoginRetriever.class);
39             retrievers.put(FunctionalUnit.HourlyLogin, retriever);
40           }
41
42         case Hourly500Requests - >{
43             log.info("Instantiating HourlyRequestRetriever");
44             context.registerBean(Hourly500RequestRetriever.class,
          Hourly500RequestRetriever::new);
45             retriever = context.getBean(Hourly500RequestRetriever.class);
46             retrievers.put(FunctionalUnit.Hourly500Requests, retriever);
47           }
48
49         default - >{
50             log.info("Unexpected functional unit: {}", functionalUnit);
51             throw new IllegalStateException("Unexpected functional unit:
          " + functionalUnit);
52           }
53         }
54       }
55       log.info("Found {}", retriever.getClass().getSimpleName());
56
57       return retriever;
58     }
59   }
```

[7] ApiCallHistoryDto

```
/**
 * Represents the db entity for table API_CALL_HISTORY
 * @implNote it's a read-only entity, in fact it's not annotated with <
   code>@Table(...)</code>
 */
@Getter
@Entity(name = "API_CALL_HISTORY")
public class ApiCallHistory {@Id@GeneratedValue
  private Long id;

  @Column(name = "api_key")
  private String apiKey;

  @Column(name = "execution_date")
  private Date execution_date;

  @Column(name = "requests_counter")
  private Long requests_counter;
}
```

[8] postIngestionV2

```
public void postIngestionV2(ArrayList<ResponseDto> monitoringResponse)
    {
     log.info(FootprintServiceImpl.class.getSimpleName() + "." + Thread.
    currentThread().getStackTrace()[1].getMethodName()
             + "(ArrayList<ResponseDto> monitoringResponse)" + "posting
    monitoring metrics to {} endpoint", ingestionV2Endpoint);

     StopWatch stopWatch = new StopWatch();
     stopWatch.start("Total ingestion");
     log.info("Sending all request to IngestionV2");
     sendAllIngestionV2(monitoringResponse);
     stopWatch.stop();

     log.info(FootprintServiceImpl.class.getSimpleName() + "." + Thread.
    currentThread().getStackTrace()[1].getMethodName()
             + "(ArrayList<ResponseDto> monitoringResponse) all metrics
    posted to ingestionV2 with elapsed time: {} ms", stopWatch.
    lastTaskInfo().getTimeMillis());
}
```

[9] sendPostRequestToIngestion

```
1  private void sendPostRequestToIngestion(IngestionV2RequestDto
       requestIngestionV2) {
2      String ingestionV2Endpoint = "https://nttdata-green-calculator.com/
       gateway/v2/ingestion-event";
3      ObjectMapper mapper = new ObjectMapper();
4
5      HttpResponse<String> response;
6      try (final HttpClient client = HttpClient.newHttpClient()) {
7          String requestBody = mapper.writeValueAsString(
       requestIngestionV2);
8          HttpRequest request = HttpRequest.newBuilder()
9                  .uri(URI.create(ingestionV2Endpoint))
10                 .header("x-api-key", adapterProperties.apikey())
11                 .header("x-token", adapterProperties.apikey())
12                 .header("Content-Type", "application/json")
13                 .POST(HttpRequest.BodyPublishers.ofString(requestBody))
14                 .build();
15
16         response = client.send(request, HttpResponse.BodyHandlers.
       ofString());
17     }
18     if (response.statusCode() != 200) throw new HttpResponseException(
       response.statusCode(), response.body());
19 }
```

Esempi request body

Tabella 5.1: Esempio di request body per /v1/embodiedEmissions

Parametro	**Valore**
type	RACK
nCPUs	10
memorySize	32 GB
nSSD	2
nHDD	3
nGPU	2

Tabella 5.2: Esempio di request body per /v1/carbonfootprint/bulk/serverless/co2

Parametro	**Valore**
reqID	1
startDate	2023-01-18T11:00:00
endDate	2023-01-18T12:00:00
carbonIndex	ITA
PUE	GCP
cpuInfo.tdp	95 W
cpuInfo.totCores	16
cpuInfo.vCPURatio	3
cpuInfo.cpuModel	Intel Xeon Platinum 8175M
nVirtualCore	2
cpuLoad	1
gpuTdp.value	70 W
gpuTdp.gpuModel	NVIDIA T4
gpuLoad	0.7
nGPUs	3
memorySize	16 GB
storageSize	128 GB
storageType	SSD

Tabella 5.3: Esempio di request body per /v2/ingestion-event

Parametro	**Valore**
tenantId	263
applicationId	Dashboard
layerId	CLOUD_RUN
startDate	2023-01-18T11:00:00
endDate	2023-01-18T12:00:00
carbonIndex	ITA
PUE	gcp
virtualized	false
cpuTdp.value	95 W
cpuTdp.cpuModel	Intel Xeon Platinum 8175M
cpuLoad	0.7
nCores	3
totCores	4
vCPURatio	1
nVirtualCore	1
gpuTdp.value	70 W
gpuTdp.gpuModel	NVIDIA T4
gpuLoad	0.7
nGPUs	3
memorySize	16 GB
storageSize	128 GB
storageType	SSD
type	RACK
functionalUnit	
functionalUnitDescription	
totalFunctionalUnit	3
totalEmbodiedEmissions	406
lifespan	4
commissionDate	2024-05-07T12:00:00.605Z
scope	scope3