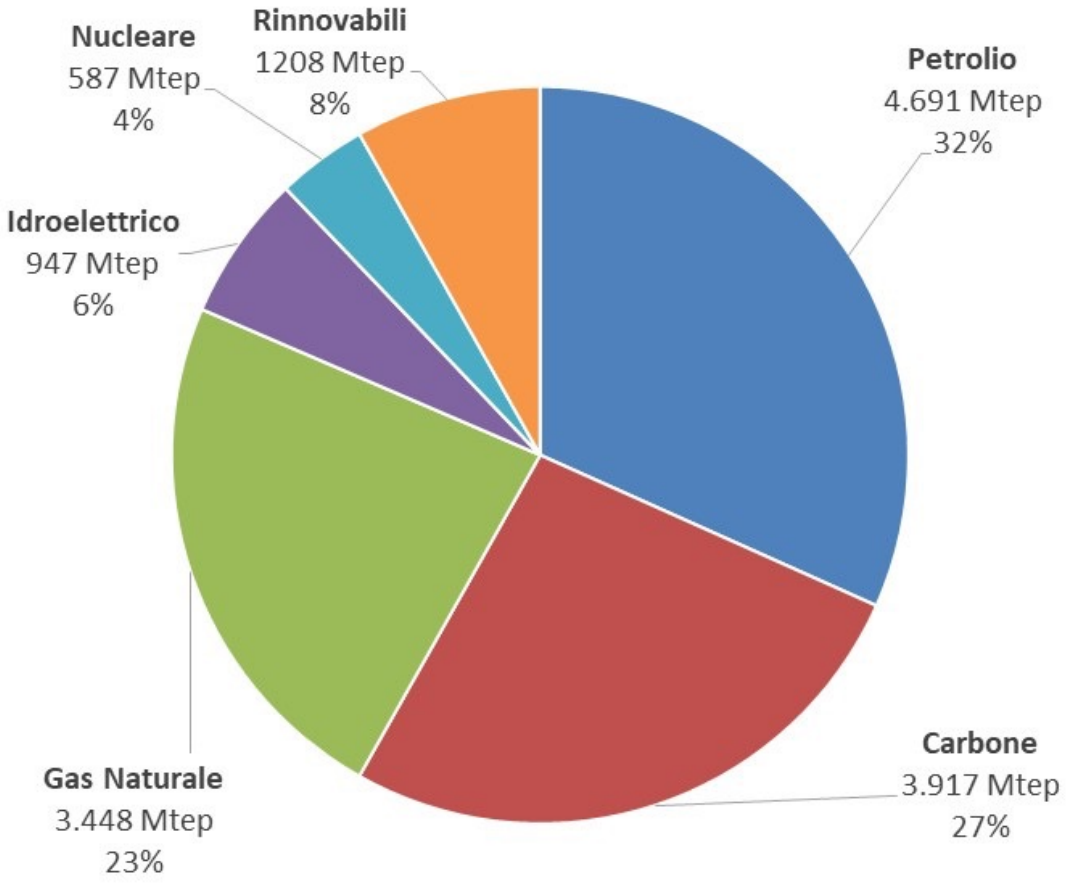
Linux Day 2024

Scegliere i linguaggi di programmazione per un futuro energeticamente sostenibile

Ing. Giovanni Alberto Vacanti

Il paniere energetico 2023 – Mondo (14.797 Mtep, +2% anno su anno)

Fonte: EI Statistical Review of World Energy 2024

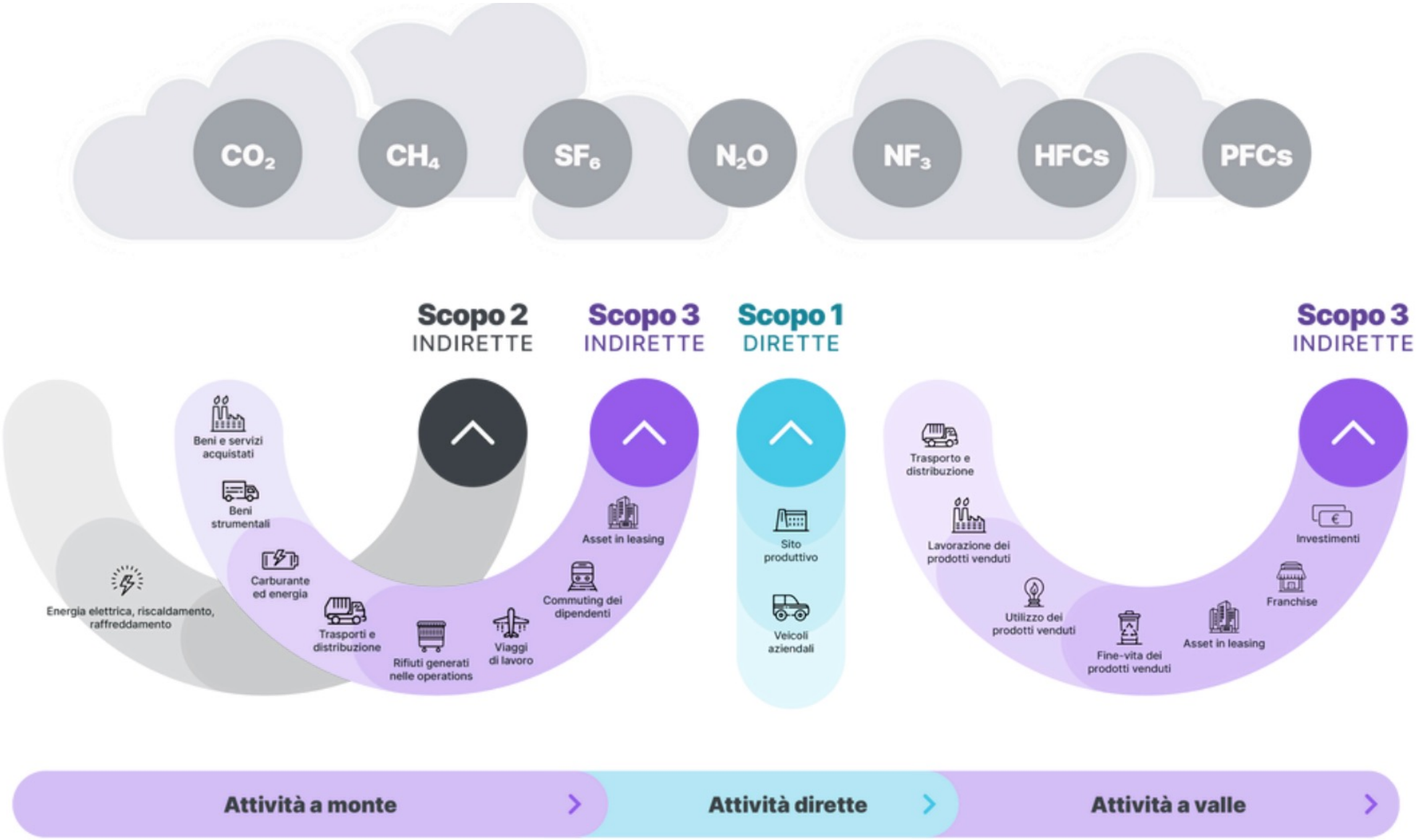


CEr
CENTRO EUROPA RICERCHE

Emissioni scope 1,2 e 3

Attualmente, la maggior parte delle aziende si concentra solo sulle proprie emissioni dirette, conosciute come emissioni **Scope 1 e 2** secondo il **Protocollo GHG** (**Gas Serra**).

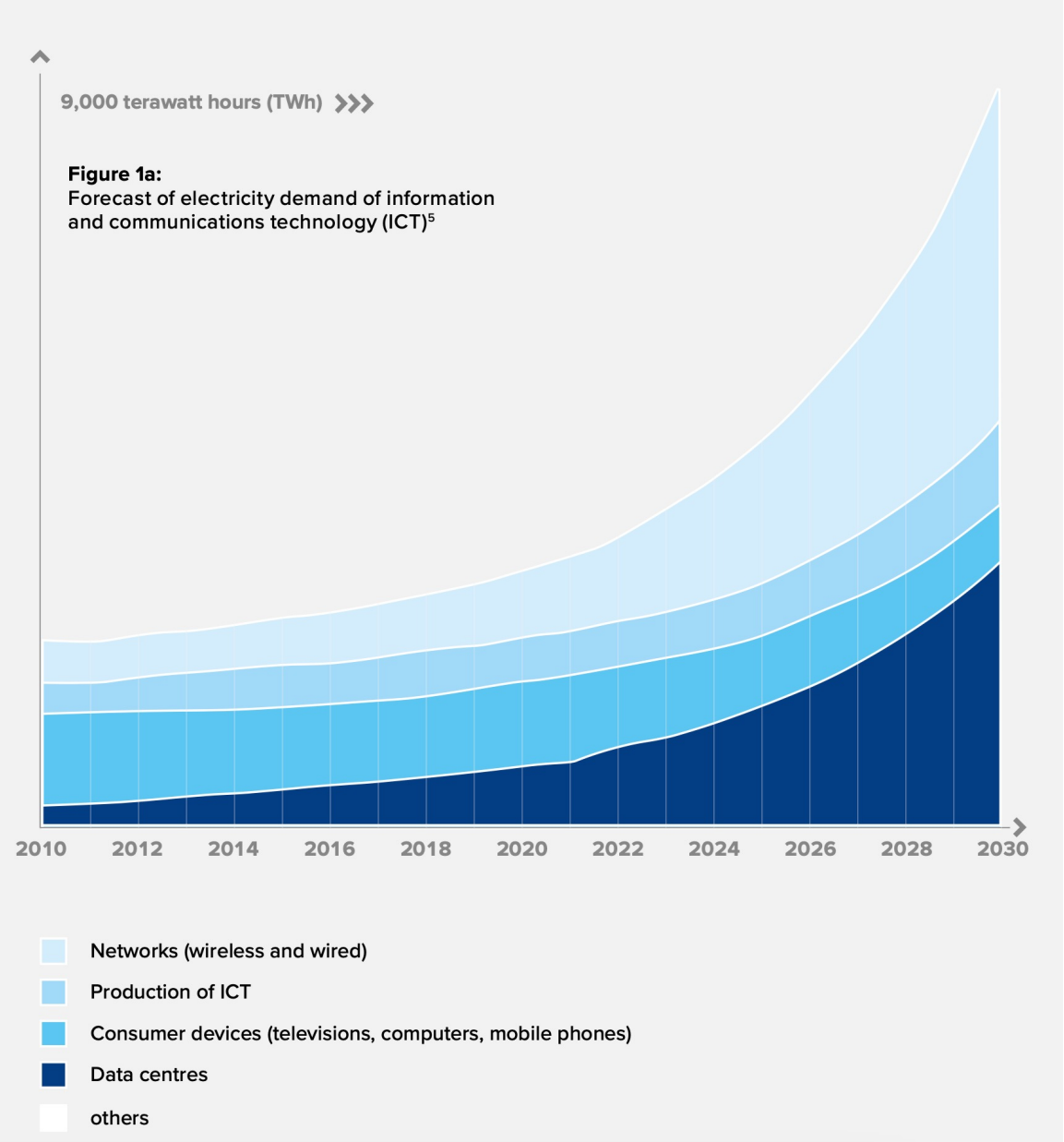
Tuttavia, per molte organizzazioni, il maggiore impatto deriva **dalle emissioni indirette (Scope 3)**, conosciute anche come emissioni della **catena del valore**, come quelle legate all'effettivo utilizzo dei prodotti.


CO_2
CH_4
SF_6
N_2O
NF_3
HFCs
PFCs
Scopo 2
INDIRETTE
Scopo 3
INDIRETTE
Scopo 1
DIRETTE
Scopo 3
INDIRETTE
Beni e servizi acquistati
Beni strumentali
Carburante ed energia
Energia elettrica, riscaldamento, raffreddamento
Trasporti e distribuzione
Rifiuti generati nelle operations
Viaggi di lavoro
Asset in leasing
Commuting dei dipendenti
Sito produttivo
Veicoli aziendali
Trasporto e distribuzione
Lavorazione dei prodotti venduti
Utilizzo dei prodotti venduti
Fine-vita dei prodotti venduti
Asset in leasing
Franchise
Investimenti
Attività a monte
Attività dirette
Attività a valle

Domanda di elettricità

La domanda di elettricità nei settori **dell'informazione** e della **comunicazione** attualmente rappresenta tra **il 5% e il 9%** della domanda globale di elettricità.

Questo numero potrebbe salire fino al **21%** entro il 2030.



Scarsa consapevolezza

C’è poca **consapevolezza pubblica**

E’ necessario sensibilizzare però **tutte** le parti coinvolte:

- Aziende
- Fornitori
- Consumatori
- Creatori di contenuti digitali

Jeremy Wagner

Alex Russell

Sfide aziendali e gestionali

Disaccoppiamento dei budget

Budget di sviluppo / budget di esercizio

C’è un conflitto di interessi:

Test di efficienza energetica
(costa farli)

vs.

Costi operativi
(aumentano se il lavoro è poco efficiente)

Dalle priorità tradizionali all'ottimizzazione del software end-to-end

Sostenibilità come **obiettivo**

Superare le **priorità tradizionali**:

- Velocità di consegna
- Riduzione dei costi

L'ottimizzazione va invece fatta lungo **l'intero ciclo di vita del software**

Occorre un **coinvolgimento** aziendale globale di:

- Manager
- Analisti
- Sviluppatori

Codice e impronta di carbonio

Ogni linea di codice che scriviamo genera **un'impronta ecologica**

Da qui la nostra responsabilità nella **riduzione dell'impatto**

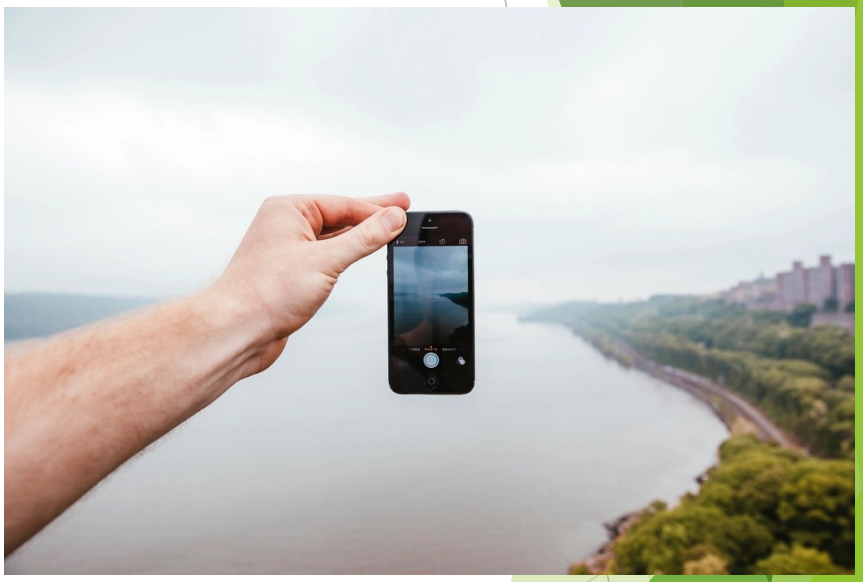
Con cloud e server residenti su infrastrutture operative 24/7, c'è un **enorme potenziale** di riduzione del consumo energetico

Non solo server…

Non solo applicazioni convenzionali o server

Desktop, smartphone, tablet, dispositivi IOT, IOE etc

Con **miliardi** di dispositivi in gioco ogni singolo pezzo di codice ha un ruolo **critico** nel poter ridurre in modo significativo le emissioni complessive di CO2



Un piccolo esempio…

App di home banking utilizzata da **5000.000 clienti**.

Sostituendo le immagini dello schermo di caricamento e riducendo le loro risoluzioni, i tempi di apertura si riducono anche di un solo millisecondo.

Apertura: 1 volta al giorno,

Risparmio: 500 ore (o più di 20 giorni!) di **tempo operativo** sui dispositivi mobili all'anno.

Diventano 50 anni se riuscissimo a risparmiare 1 secondo?

Ambito e potenziale del GreenCoding

Programmare, **distribuire** ed **eseguire** software in modo ecologicamente sostenibile

Evitare schemi di pensiero **automatici**.

Promuovere un approccio **olistico** ai problemi aziendali

Scegliere (se possible) il **linguaggio di programmazione** con maggiore efficienza energetica

Fasi del GreenCoding

1. **Pianificazione del progetto**
    a. Analisi dei requisiti iniziali
    b. Selezione della **piattaforma** e **dell'infrastruttura**

2. **Scelta del linguaggio di programmazione**
    a. **Può avere un forte impatto sull'efficienza energetica** e sulle prestazioni
    b. In alcuni casi però, altre decisioni possono avere un **impatto maggiore**

Domanda chiave del Green Coding

Il Green Coding aggiunge una **nuova domanda** al processo di progettazione:

Esiste un modo per fornire lo stesso beneficio desiderato con il minor consumo energetico?

Infrastruttura serverless:

- **Ottimizza** i consumi dell'infrastruttura

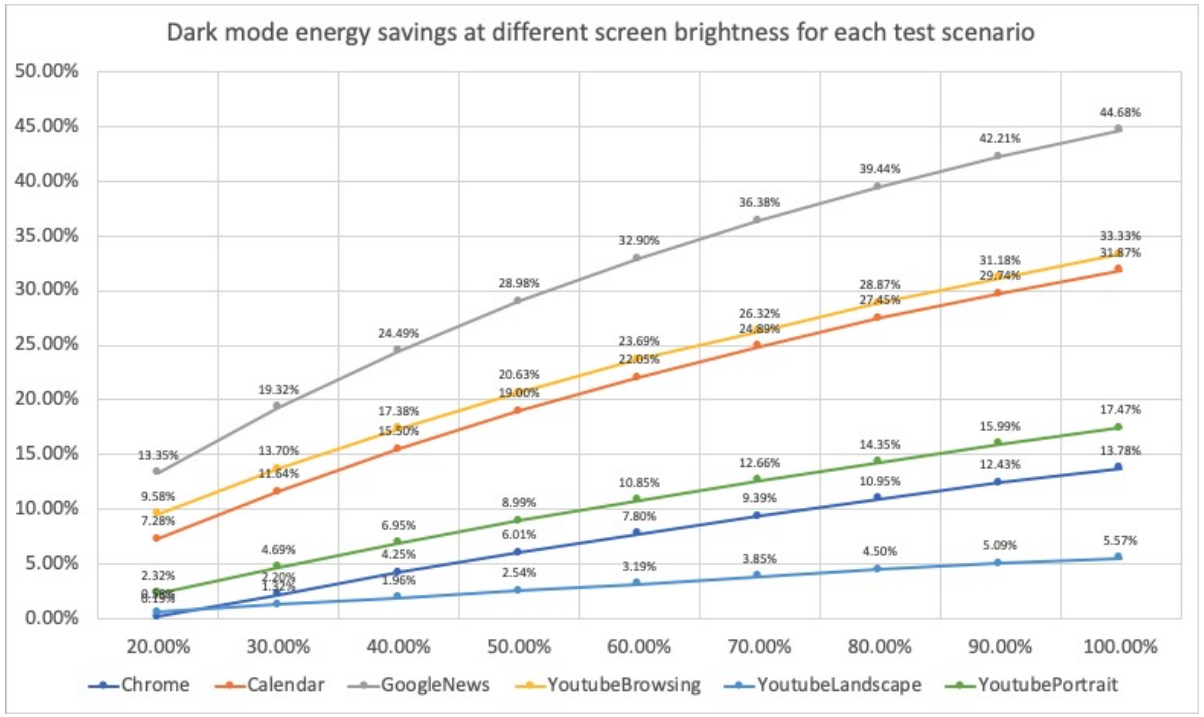
Esperienza utente:

- **Ridurre** il tempo necessario agli utenti per completare operazioni
- Ottimizzare **usabilità** e **performance** con approccio sostenibile

Il **DARK** mode

Offrire, magari di default, il **dark mode** nelle app / web app

Grazie alla tecnologia dei display OLED, il dark mode può ridurre il consumo della batteria fino al **a quasi il 24%.**



What is the impact of Dark Mode on battery drain

I pilastri della scrittura del software Green

“COSA" viene generato (il codice stesso).
È efficiente in termini di beneficio fornito o energia investita?
UNA LOGICA PIÙ GREEN

"COME" viene generato.
Il ciclo di vita dello sviluppo software è efficiente? Lo stesso codice potrebbe essere generato con meno energia?
UNA METODOLOGIA PIÙ GREEN

"DOVE" il software viene eseguito (la piattaforma finale che esegue il codice).
Consuma l’energia minima necessaria per eseguire il codice generato?
UNA PIATTAFORMA PIÙ GREEN

Principi base del risparmio

- Spegnere le luci quando non servono
- Evitare il consumo impulsivo: real time sempre ?

Spegnimento automatico

Con i servizi servless di AWS ad esempio è possible avere una riduzione dell'energia: dal momento che **AWS Lambda**, un servizio serverless di Amazon Web Service, utilizza risorse solo quando necessario, si riduce il consumo energetico rispetto a server sempre attivi.

Anche con **Google Cloud Functions** è possible lo **spegnimento automatico:** ad esempio su una piattaforma di e-commerce, se non ci sono ordini o pagamenti da elaborare, le funzioni restano **inattive**, riducendo al minimo il consumo di risorse.

Da dinamico a statico

Un altro esempio è quello della **landing page** di un'applicazione web.

Si può ricostruire la pagina **una volta e NON ogni volta.**

Una volta: per ora, per giorno, per settimana o addirittura per rilascio.

Quante persone utilizzeranno il software?

Ma anche:

Chi lo utilizzerà?

Un essere umano o un’altra «macchina», da altri sistemi ?

Con quale frequenza ?

Saltuaramente ? Ogni giorno ? Oppure ogni minuto ?

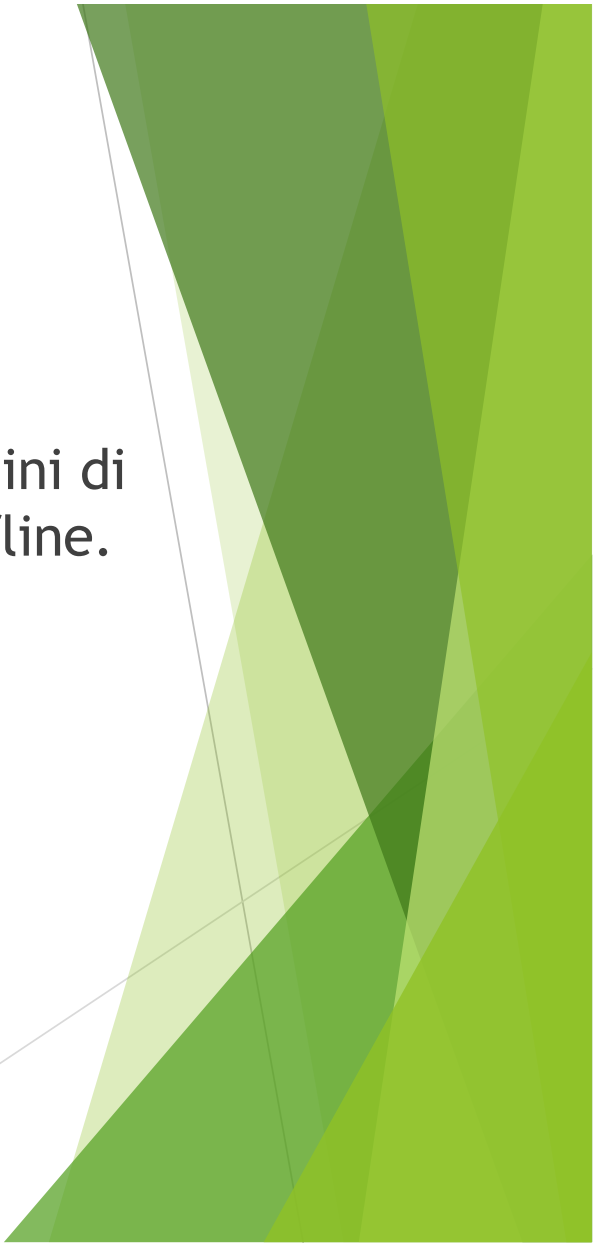
Prima di pensare al codice: come sono organizzate le risorse?

- Excel in csv
- XML in YAML
- Immagini: raster o vettoriali ? Complessità/ dimensioni

Progressive Web App (PWA)

I browser moderni possono trasformare le pagine web compatibili con gli standard **PWA** in applicazioni.

Questa tecnica offre capacità logiche più elaborate in termini di gestione della scadenza dei contenuti, oltre al supporto offline.



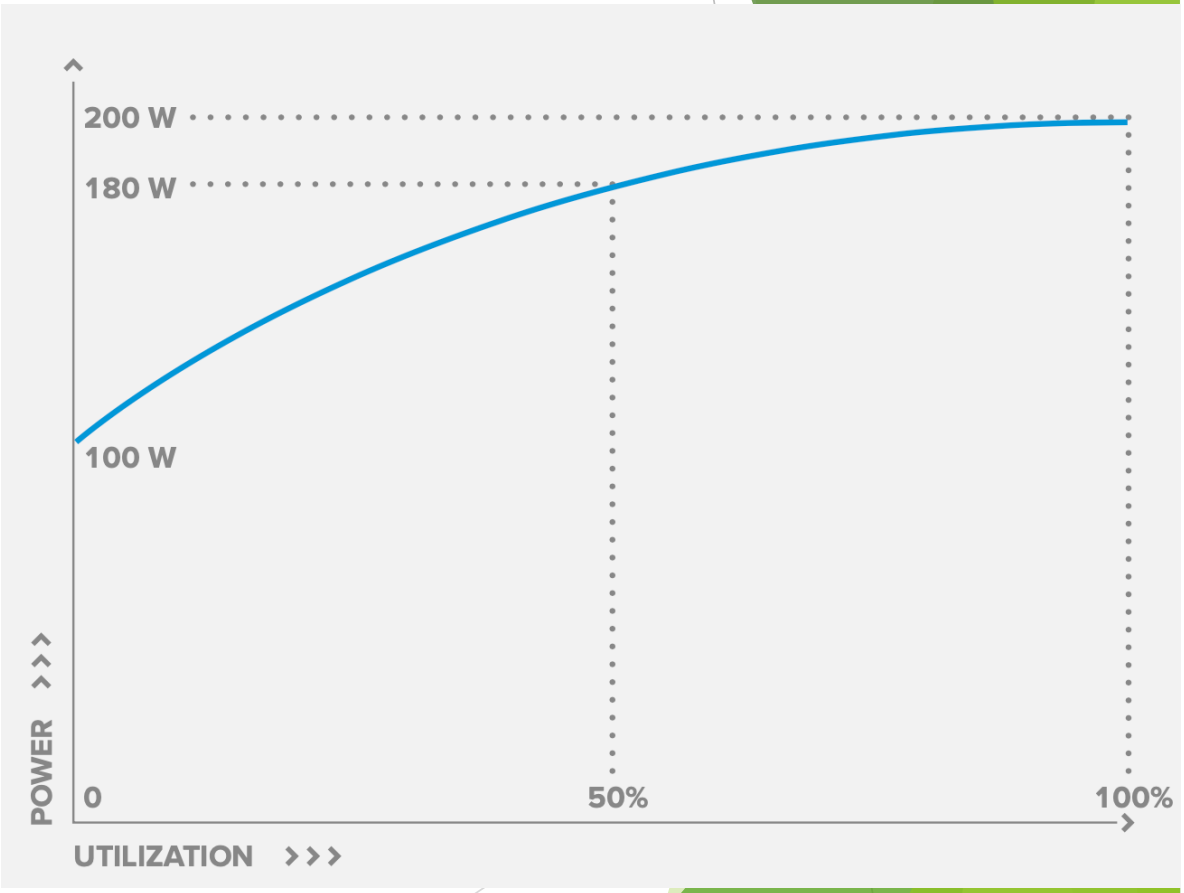
Content Delivery Network (CDN) in posizioni edge

Un **CDN** è una piattaforma altamente distribuita di server che aiuta a ridurre i ritardi nel caricamento dei contenuti delle pagine web **riducendo la distanza** fisica tra il server e l'utente.

Utilizzazione ottimale

L'energia consumata da un sistema informatico **non è proporzionale** ai livelli di utilizzo.

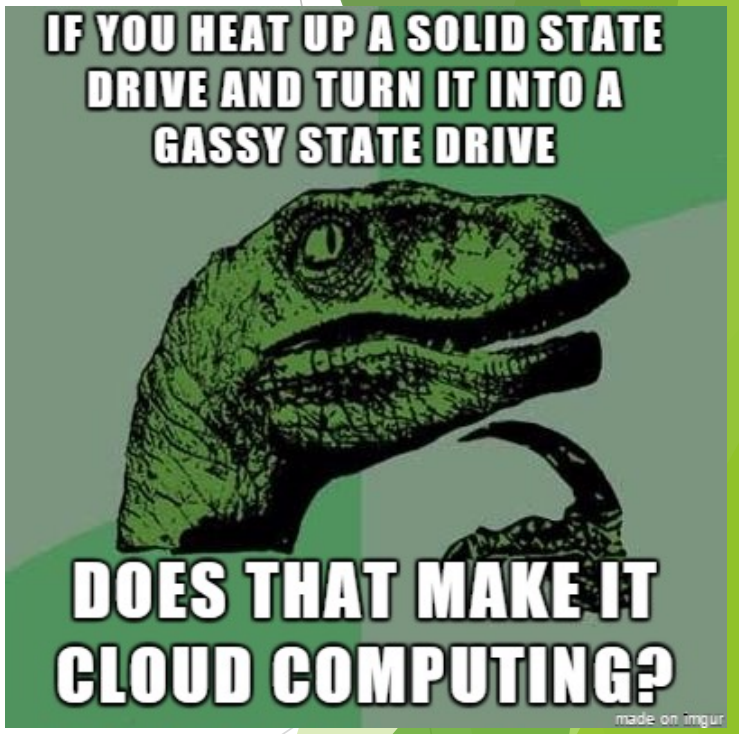
Questo concetto è noto come **proporzionalità energetica**: all'aumentare dei livelli di utilizzo, il **consumo energetico** per ogni punto percentuale di potenza di calcolo utilizzata **diminuisce**.



Il Cloud

Poiché la maggior parte dei sistemi esegue più applicazioni **contemporaneamente**, diventa difficile identificare il consumo energetico di una singola applicazione, come ad esempio un programma che gira su un monolite condiviso insieme a numerose altre applicazioni.

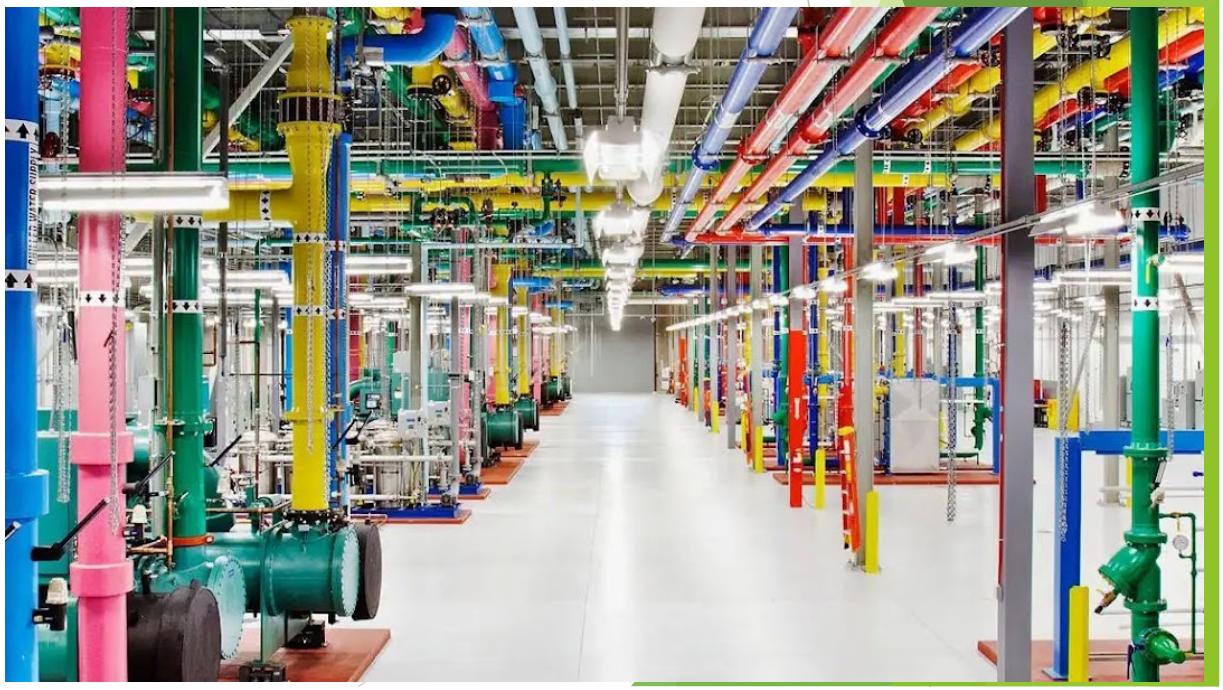
Con il cloud, invece, le correlazioni sono più chiare: **più alta è la fattura, più energia è stata consumata**.



L’esempio di Google Cloud

Google Cloud ha fatto un ulteriore passo avanti, utilizzando il **machine learning** per ridurre fino al **40% l’energia necessaria al raffreddamento.**

Google Cloud



Rapporto NRDC (**Natural Resources Defense Council**)

Il **tasso di utilizzo dei server** nei grandi provider come **AWS, Google Cloud e Microsoft Azure** si attesta intorno al **65%.**

Mentre i data center **on-premise** tradizionali operano con un livello di utilizzo compreso tra **12% e 18%.**

Scelta del linguaggio

La scelta del linguaggio di programmazione può **influenzare** l'impronta di **CO2** di un software principalmente per via di come gestisce l'efficienza dell'elaborazione e dell'hardware.

Linguaggi «vicini» all’hardware

Alcuni linguaggi sono più vicini **all'hardware** (come il C) consentendo di scrivere **codice più efficiente** che sfrutta meglio le risorse del computer (**CPU**, **memoria**), riducendo così il tempo di elaborazione.

Compilazione vs Interpretazione

I **linguaggi compilati** (come C,Go o Rust) tendono a produrre codice eseguibile **più efficiente**

Linguaggi interpretati (come Python o JavaScript) richiedono l'intervento di un interprete **ad ogni esecuzione**, aumentando il carico computazionale e il consumo di energia.

Total			
Language	Energy	Language	Time
C	1.00	C	1.00
Rust	1.03	Rust	1.04
C++	1.34	C++	1.56
Java	1.98	Java	1.89
Pascal	2.14	Go	2.83
Lisp	2.27	Pascal	3.02
Swift	2.79	C#	3.14
C#	3.14	Lisp	3.40
Go	3.23	Swift	4.20
JavaScript	4.45	JavaScript	6.52
TypeScript	21.50	PHP	27.64
PHP	29.30	TypeScript	46.20
Ruby	69.91	Ruby	59.34
Python	75.88	Python	71.90

Programming Language	Energy Consumption (J)	Speed of Execution (ms)
C	57	2,019
Rust	59	2,103
C++	77	3,155
Ada	98	3,740
Java	114	3,821

Programming Language	Energy Consumption (J)	Speed of Execution (ms)
Perl	4,604	132,856
Python	4,390	145,178
Ruby	4,045	119,832
Lua	2,660	167,416

Source: SLE 2017: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering

Tempi di esecuzione

Linguaggi con overhead significativo (come **Python** per operazioni intensive) possono richiedere più tempo per eseguire lo stesso compito rispetto a linguaggi più performanti, aumentando il **consumo energetico totale**.

Total			
Language	**Energy**	**Language**	**Time**
C	1.00	C	1.00
Rust	1.03	Rust	1.04
C++	1.34	C++	1.56
Java	1.98	Java	1.89
Pascal	2.14	Go	2.83
Lisp	2.27	Pascal	3.02
Swift	2.79	C#	3.14
C#	3.14	Lisp	3.40
Go	3.23	Swift	4.20
JavaScript	4.45	JavaScript	6.52
TypeScript	21.50	PHP	27.64
PHP	29.30	TypeScript	46.20
Ruby	69.91	Ruby	59.34
Python	75.88	Python	71.90

Misurare il consumo energetico del sistema

Il primo passo è determinare quanto **energia elettrica** il software sta utilizzando.

Questa fase richiede la **raccolta di dati** relativi al consumo di energia della **CPU**, della **GPU** e di altre risorse del sistema.

Metodi per misurare il consumo energetico

Utilizzo diretto di strumenti di monitoraggio hardware: è possibile **monitorare direttamente** il consumo energetico dell'hardware tramite strumenti integrati nel sistema operativo o software esterni.

Strumenti utili

Linux

E’ possible usare comandi come **powerstat** o **powertop** per misurare il consumo energetico del sistema.
Questi strumenti mostrano la potenza utilizzata (in watt) dai componenti hardware (CPU, GPU, dischi).

MacOS

Il tool **powermetrics** monitora il consumo energetico della CPU e del sistema.

Windows

Strumenti come **Intel Power Gadget** o **Windows Performance Monitor** consentono di monitorare l'energia consumata dalla CPU e altri componenti.

Strumenti utili

Dispositivi hardware

E’ possible usare anche **dispositivi hardware**, come prese intelligenti o wattmetri, per misurare **direttamente** il consumo energetico del computer su cui gira il software.

API e strumenti specifici

Intel RAPL (Running Average Power Limit) su CPU Intel, che consente di stimare il consumo di energia per core.

NVIDIA Management Library (NVML), per monitorare il consumo di energia delle GPU NVIDIA.

Calcolo del consumo del software

- Misurare il consumo del sistema **a riposo** (senza il software in esecuzione).
- Misurare il consumo del sistema **con il software in esecuzione**.
- La differenza tra i due fornisce una **stima** del consumo energetico del software.

Stima delle emissioni di CO_2

Una volta ottenuto il consumo energetico del software (espresso in **kWh**), è possibile stimare le emissioni di CO_2 usando un **fattore di conversione** che dipende dalla fonte energetica utilizzata.

Formula generale:

$$\text{Emissioni } CO_2(kg) = \text{Consumo energetico}(kWh) \times \text{Fattore di emissione } CO_2(kg/kWh)$$

Fattore dei emissione

Il **fattore di emissione CO2** varia in base al mix energetico del Paese o dell'area geografica.

Per esempio:

- In un paese che utilizza principalmente **carbone**, il fattore di emissione potrebbe essere intorno a **0.9 kgCO2/kWh**.
- In paesi che utilizzano più **energia rinnovabile**, il fattore potrebbe essere più basso, ad esempio **0.05 kgCO2/kWh**.

Esempi di fattori di emissione per alcuni paesi

Stati Uniti: Circa 0.4-0.5 kgCO2/kWh.

Unione Europea: Circa 0.25 kgCO2/kWh.

Francia: Circa 0.06 kgCO2/kWh (grazie all'uso dell'energia nucleare).

Germania: Circa 0.45 kgCO2/kWh (dipende ancora dal carbone).

Misura in ambienti Cloud

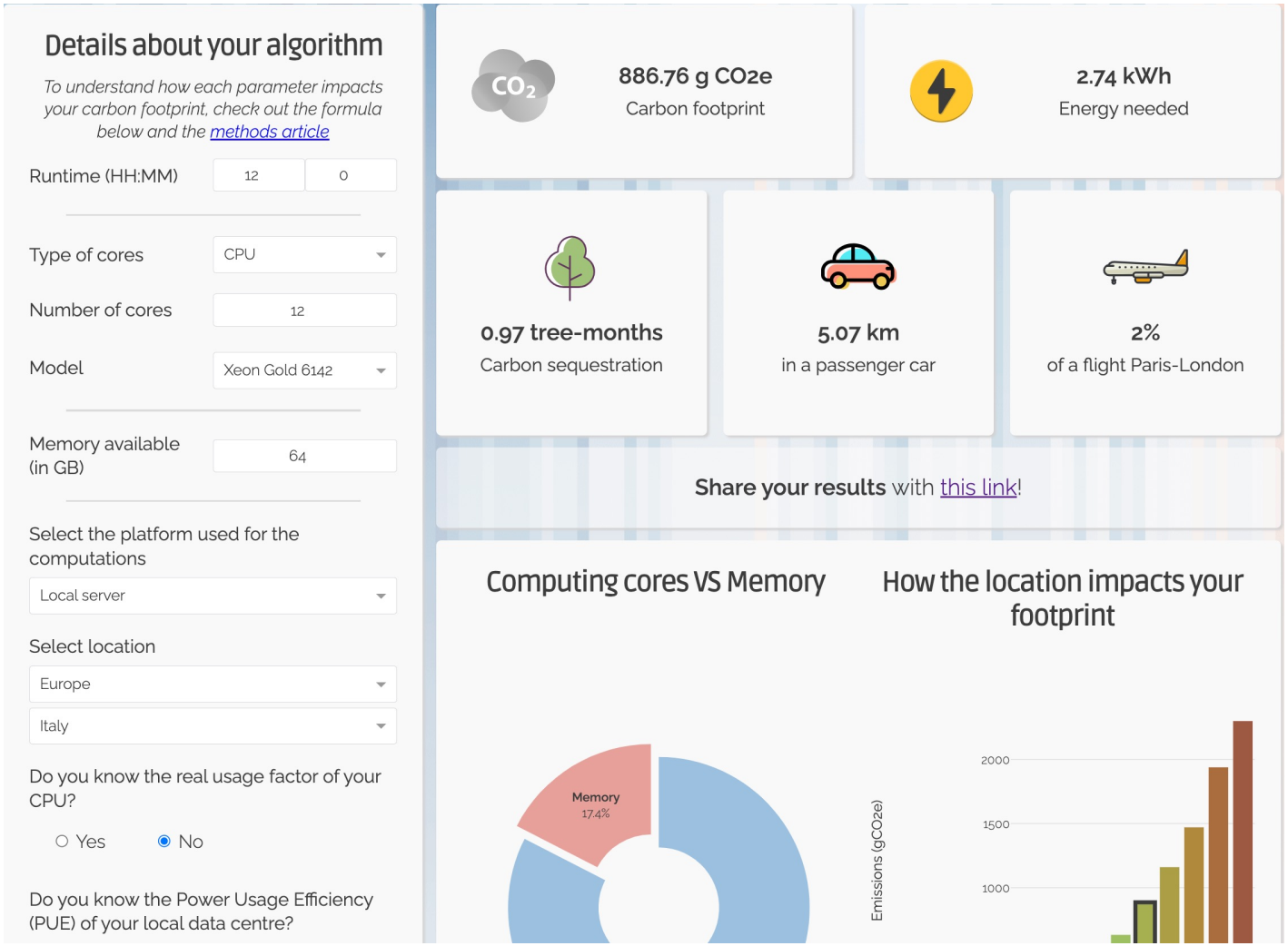
Cloud Carbon Footprint: Se il software gira su una piattaforma cloud (AWS, GCP, Azure), strumenti come **Cloud Carbon Footprint** possono **stimare** le emissioni in base al consumo delle risorse cloud utilizzate (CPU, memoria, traffico di rete) vedi https://www.cloudcarbonfootprint.org/

Green Algorithms: È uno strumento open-source che permette di stimare il consumo di energia e le emissioni di CO2 basate sul tempo di esecuzione e sulle risorse hardware (utilizzato in particolare per algoritmi di intelligenza artificiale) vedi https://calculator.green-algorithms.org/

Scaphandre: Uno strumento **open-source** che misura il consumo di energia dei server e dei container, utile in ambienti cloud e distribuiti.

Sustainable Computing Consortium: Alcune organizzazioni forniscono linee guida e strumenti per monitorare e ottimizzare l'efficienza energetica del software.

https://calculator.green-algorithms.org/



Misurare l’impatto energetico di query SQL o database

- Misurare il consumo energetico del server database **prima e dopo** l’esecuzione delle query.
- Identificare le query SQL più costose in termini di **risorse**, come query che richiedono scansioni su tabelle grandi o indici non ottimizzati.

Calcolo dell’impronta su dispositivi mobili

Per applicazioni su dispositivi mobili, strumenti come **Android Profiler** e **Xcode Instruments** permettono di monitorare il consumo energetico delle app.

Utilizzando questi dati, è possibile stimare l’impatto in termini di CO2, seguendo lo stesso processo di conversione basato sull'energia.

Ottimizzazione

Ottimizzazione delle risorse: Oltre a **misurare**, è importante **ottimizzare il codice** per ridurre il consumo energetico, ad esempio **ottimizzando le query SQL** o migliorando **l'efficienza degli algoritmi**, riducendo in tal modo il carico computazionale.

Analisi temporale: Si possono effettuare **analisi nel tempo** per vedere se le ottimizzazioni nel codice hanno portato a una riduzione nel consumo di energia e quindi nelle emissioni di CO2.

Cosa ottimizzare ?

- **Ottimizzazione del runtime (Rust o Java)**
- **Sfruttamento del parallelismo (Go o Erlang)**
- **Fare attenzione a librerie e Framework**
- **Ottimizzazione degli algoritmi** O(n log n) è meglio di O(n^2)
- **Riduzione del carico computazionale**
  - **Eliminare calcoli ridondanti**
  - **Lazy Evaluation**

Cosa ottimizzare ?

- Gestione della memoria efficiente
- Programmazione asincrona e parallela
- Minimizzazione delle operazioni di I/O
- Batching e caching
- Rilasciare risorse non necessarie
- Profilazione e ottimizzazione
- Utilizzare la programmazione dichiarativa o funzionale

Ottimizzazione delle query

L'ottimizzazione delle **query SQL** riduce il numero di operazioni computazionali e il tempo necessario per eseguirle, abbassando il consumo energetico.

Evitare SELECT *

Ovvero evitare di selezionare tutte le colonne, se non necessario.

Selezionare solo le colonne richieste.

-- **Inefficiente**

- SELECT * FROM utenti WHERE id = 1;

-- **Efficiente**

- SELECT nome, cognome FROM utenti WHERE id = 1;

Qualche consiglio…

- Indicizzare correttamente
- Usare le query con LIMIT
- Evitare subquery annidate
- Indici sui campi giusti
- Usare indici composti
- Mantenere piccoli gli indici (indici parziali)
- Partizionamento delle tabelle
- Bilanciare normalizzazione e denormalizzazione
- Caching e buffering
- Gestione dell'hardware (SSD/compressione/ottimizzazione del carico)
- Eliminare o archiviare dati obsolete
- Profilazione e tuning continuo

Json

Anche l'uso di **JSON** (JavaScript Object Notation) può essere ottimizzato per migliorare l'efficienza energetica, soprattutto quando si lavora con database, API, o trasferimenti di dati in rete.

Ottimizzare JSON significa ridurre la dimensione dei dati trasferiti, migliorare il parsing e diminuire la complessità computazionale, riducendo il consumo di risorse CPU, memoria e larghezza di banda.

Qualche consiglio…

- **Ridurre la dimensione dei payload JSON**
  - Eliminare i dati non necessari
  - Usare formati abbreviati per i campi
  - Compattare JSON
- **Compressione dei dati JSON**
- **Paginare i dati JSON**

Strutturare i dati in modo efficiente

Utilizzare array anziché oggetti quando appropriato: Gli array sono più efficienti degli oggetti in termini di parsing e uso di memoria, specialmente quando si tratta di una collezione di elementi simili.

```
// Inefficiente (uso di oggetti con chiavi ridondanti)
 { "utenti":
       {
       "1": { "nome": "Mario", "cognome": "Rossi" },
       "2": { "nome": "Luigi", "cognome": "Bianchi" }
       }
 }
// Migliore (uso di array)
{ "utenti": [
       { "id": 1, "nome": "Mario", "cognome": "Rossi" },
       { "id": 2, "nome": "Luigi", "cognome": "Bianchi" }
                ]
}
```

Streaming di JSON

Quando si lavorano grandi quantità di dati in JSON (ad esempio, in un database o in API RESTful), usare la tecnica di **streaming** per elaborare i dati **man mano che vengono letti**, senza doverli caricare interamente in memoria.

Questo riduce il consumo di memoria e migliora l'efficienza energetica.

- In MySQL, ad esempio, si può utilizzare **JSON streaming** con l'API nativa di JSON per processare grandi set di dati senza sovraccaricare la memoria.
- In linguaggi come **Java** o **Node.js**, librerie come **Jackson Streaming API** o **JSONStream** permettono di gestire grandi file JSON in modalità stream, evitando il carico di memoria durante il parsing.

Ottimizzazione del parsing JSON

Ridurre l’overhead associato alla deserializzazione e serializzazione JSON.

Usare parser JSON efficienti: Utilizzare librerie JSON ottimizzate per performance (come **Jackson** per Java o **FastJSON** per Python) può ridurre il tempo di elaborazione e quindi il consumo energetico.

Precompilazione di schema: Usare uno schema JSON predefinito per verificare e convalidare i dati riduce il tempo di parsing, poiché consente di saltare passaggi non necessari nel processo di validazione dinamica.

Minimizzare il numero di richieste JSON

Quando possibile, ridurre il numero di richieste JSON migliorando l'aggregazione di dati.

Batching delle richieste: Inviare **più richieste** JSON in batch piuttosto che **singole richieste** separate riduce la latenza e la quantità di risorse utilizzate per ogni richiesta.json

```
{ "richieste":
    [
        { "utente_id": 1, "azione": "getDettagli" },
        { "utente_id": 2, "azione": "getDettagli" }
    ]
}
```

Design patterm

Nella programmazione orientata agli oggetti (OOP), alcuni design pattern possono aiutare a ottimizzare il consumo energetico del software, riducendo il carico computazionale e migliorando l'efficienza delle risorse.

L'obiettivo di questi pattern è **ridurre il numero di operazioni** costose, **evitare sprechi di memoria** e **calcoli ridondanti,** e migliorare l'uso della CPU, della RAM e delle risorse I/O.

Alcuni pattern green

- Lazy Initialization / Lazy Loading
- Flyweight
- Object Pool
- Singleton
- Proxy
- Builder
- Observer

Proxy Pattern

Il **Proxy Pattern** è un design pattern strutturale utilizzato nella programmazione orientata agli oggetti che fornisce un oggetto **surrogate** o placeholder per controllare l'accesso a un altro oggetto.

Come riduce il consumo energetico ?

Evita l'accesso a risorse costose in termini di energia, come una connessione di rete, a meno che non sia effettivamente richiesto.

Obiettivi principali

- **Controllo di accesso**: Un proxy può **limitare** o **condizionare** l'accesso a un oggetto reale.
- **Lazy Initialization**: **Ritarda** la creazione dell'oggetto reale fino a quando non è strettamente necessario.
- **Riduzione del carico**: Può agire come **cache** o **ridurre il carico computazionale** delegando le operazioni solo quando richieste.
- **Protezione**: Un proxy può verificare i permessi prima di consentire l'accesso all'oggetto reale.

```
interface Immagine {
void visualizza();
}
class ImmagineReale implements Immagine {
private String file;
public ImmagineReale(String file) {
        caricaImmagineDaDisco(file);
        // Operazione costosa
    }
private void caricaImmagineDaDisco(String file) {
        System.out.println("Caricamento dell'immagine
" + file);
    }
public void visualizza() {
        System.out.println("Visualizzazione
dell'immagine " + file);
    }
}
```

```
class ProxyImmagine implements Immagine {
    private String file;
    private ImmagineReale immagineReale;
    public ProxyImmagine(String file) {
        this.file = file;
    }
    public void visualizza() {
        if (immagineReale == null) {
            immagineReale = new ImmagineReale(file);
            // Carica l'immagine solo se necessaria
        }
        immagineReale.visualizza();
    }
}
```