ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Scuola di Scienze
Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

Green Coding
programming languages and more

Relatore:
Prof. Ivan Lanese

Presentata da:
Alessandro Severini

Sessione unica 31 Ottobre 2025

Anno Accademico 2024/2025

Alla mia famiglia,
per il sostegno incondizionato,
soprattutto per accogliere questo strano soggetto dai capelli lunghi,
con particolare menzione ai miei genitori e a mia sorella,
riusciti nell'impresa di farmi indossare un completo.

A Sara, mia compagna di vita
e futura moglie,
per supportarmi e sopportarmi,
soprattutto quando dimentico le cose, come l'aglio nel piatto.

Ai miei amici di Castelfidardo e dintorni,
per la loro unicità tra tutti gli altri,
costretti a soffrire la lontananza della persona migliore e più umile che conoscano, ossia io.

Ai miei amici di Bologna,
sia quelli che abitano ancora qua che quelli la cui vita li ha portati altrove,
per avermi offerto una seconda famiglia,
nonostante ormai mi conoscano.

Alla città di Bologna,
che mi ha mostrato realtà uniche e variopinte,
permettendomi di guardare il mondo nuovamente con gli occhi di un bambino.

All'Alma Mater Studiorum e al corpo docenti,
per l'aria di cultura e storicità che mi hanno permesso di respirare,
con particolare menzione al mio relatore, il professor Lanese,
che ha avuto la pazienza di seguirmi, anche con i miei momenti altalenanti.

A tutto e soprattutto a tutti,
tutti coloro che non ho potuto menzionare singolarmente,
perchè tutti sono un pezzo fondamentale di questo traguardo
e della mia vita.

Grazie.

(E un grazie lo dedico anche a me stesso, per continuare a provarci)

Indice

Introduzione

Al giorno d'oggi l'informatica e la programmazione sono parte integrante delle nostre vite quotidiane sia per l'uso domestico che ne facciamo sia per quanto riguarda le attività lavorative. Da questo vasto e capillare utilizzo consegue ovviamente un corrispettivo consumo energetico, la cui crescente mole è già da anni un problema generalmente riconosciuto. Il settore è infatti in costante ed esponenziale crescita e porta ad un consumo enorme di elettricità [19] [24].
Alcuni dati raccolti durante questa survey dicono infatti che, già nel 2006, vi erano circa 6000 data centers negli Stati Uniti che consumavano 61 miliardi di kWh costituendo l'1.5% di tutto il consumo di energia elettrica nazionale, con una richiesta in crescita del 12% annuo [19].
Altri dati più recenti confermano il trend: a partire dal 2020, le emissioni legate all'informatica hanno raggiunto oltre il 2,1% delle emissioni totali mondiali, sono quindi quasi raddoppiate nell'ultimo decennio e si prevede un aumento di un ulteriore 40% entro il 2030. L'industria informatica ora emette CO_2 a un livello simile a quello dell'industria aerea, nota per il suo impatto sull'ambiente [16] [24].
Questo incremento vertiginoso di consumi deriva, ovviamente, dalla rapidissima diffusione dell'informatica inizialmente menzionata, così come dalla sua altrettanto rapida evoluzione.
Le apparecchiature e i sistemi sono sempre più potenti e complessi e lavorano su moli di dati sempre più grandi, questo richiede una potenza di elaborazione generalmente maggiore che comporta quindi un maggiore consumo energetico.
D'altronde, è l'esecuzione delle componenti hardware, comandate dal software, a generare nell'effettivo il consumo di energia che si può dunque considerare essere generata dal tempo di esecuzione e dalla potenza utilizzata.
Questa correlazione verrà analizzata dettagliatamente nel corso della survey.

Da queste consapevolezze, e con l'emergere di questi dati allarmanti, nasce il concetto di Green Computing, noto anche come Green IT o Informatica Sostenibile. Non si tratta di un singolo strumento o una pratica isolata, ma di un approccio a tutto tondo che mira a ridurre

l'impatto ambientale dell'intero ciclo di vita dei sistemi informatici. Questo include tutto, dalla progettazione e produzione dei dispositivi, fino al loro utilizzo e smaltimento.
Il concetto di Green Computing non si limita alla semplice ottimizzazione del consumo energetico, ma si estende alla promozione della sostenibilità ambientale in ogni fase. L'obiettivo è ridurre il consumo di energia e minimizzare l'impatto ambientale generale.
Questa disciplina si articola in diversi ambiti, che vanno dall'hardware al software, e coinvolge diverse figure professionali. Per esempio, nell'ambito dell'hardware, si possono considerare l'utilizzo di materiali riciclati o la progettazione di componenti più efficienti dal punto di vista energetico. Lato software, invece, l'attenzione si sposta sull'ottimizzazione del codice per renderlo più efficiente e, di conseguenza, meno energeticamente costoso.

L'adozione di pratiche di green computing è una questione sia economica che etica, infatti le aziende che dimostrano leadership ambientale e trasparenza hanno più probabilità di ottenere un vantaggio competitivo e migliorare la propria reputazione, così come possono ottenere una riduzione dei costi energetici che a volte rappresentano fino al 50% delle spese totali di un'organizzazione [3].
Tutto ciò comporta un vasto numero di cambiamenti necessari per conseguire questo obbiettivo e i vari stakeholder del settore IT dovranno adottare approcci diversi da quelli attuali, considerando la componente energetica in tutto il ciclo di vita del computer.
Le possibili ottimizzazioni spaziano per una grande varietà di livelli, molti dei quali sono già analizzati da tempo e hanno delle soluzioni più o meno definite, ci sono invece altri campi di più recente considerazione e ampio margine di crescita, quali l'efficienza energetica intrinseca di determinati linguaggi, algoritmi, strutture dati, framework e tanti altri racchiusi in una sottocategoria del Green Computing denominata Green Coding, la quale riguarda quindi gli aspetti del settore IT legati alla programmazione.
Il Green Coding sarà l'oggetto principale di questa survey, verranno comunque tenuti in considerazione anche elementi del Green Computing che si interfacciano e collaborano inevitabilmente con quelli del Green Coding.

Date le premesse è naturale quindi porsi alcune domande:

1. È necessario adottare Green Coding?

2. Cosa influenza l'efficienza energetica di un programma?

3. Esistono linguaggi migliori di altri?

4. Esistono configurazioni e implementazioni più efficienti?
5. Quali algoritmi, strutture dati e tecniche di programmazione risultano in rilievo?
6. Quali strumenti consentono uno sviluppo più green?
7. Cosa si può dire del contesto più ampio? Ossia del Green Computing
8. Si possono definire delle linee guida dai dati attualmente esistenti?

In questa survey esamineremo dapprima il contesto del Green Coding e cosa esso comprende, andremo poi a guardare cosa effettivamente influenza l'efficienza energetica analizzando le problematiche esistenti e alcune soluzioni possibili.
Infine si cerca di determinare delle linee guida da seguire tenendo in considerazione il contesto, l'ambiente d'esecuzione, il linguaggio di programmazione, strumenti a supporto e metodologie di Green Coding come risultato di un'elaborazione delle informazioni raccolte, tenendo in considerazione anche il contesto più ampio del Green Computing.

1. È necessario adottare Green Coding?

Come abbiamo visto nell'introduzione, il settore dell'informatica ha un impatto ambientale significativo e in continua crescita. Questo problema non si risolve semplicemente con l'uso di hardware più efficiente o con data center alimentati da energie rinnovabili. Dopotutto, l'hardware e i data center non lavorano da soli: il software fa eseguire i calcoli, fa elaborare i dati e comanda le operazioni ed è proprio l'inefficienza del software a rappresentare una delle maggiori fonti di spreco energetico. È qui che entra in gioco il Green Coding.

Il Green Coding è una branca del Green Computing che si concentra specificamente sull'ottimizzazione del codice e delle pratiche di sviluppo per ridurre il consumo di energia. Non si tratta semplicemente di scrivere codice che funziona, ma di scrivere codice che funzioni nel modo più efficiente possibile, consumando meno risorse.
Per capire l'importanza del Green Coding, pensiamo a un algoritmo: se questo è scritto in modo inefficiente, potrebbe richiedere migliaia o persino milioni di operazioni in più per completare un compito. Ogni singola operazione consuma una piccola quantità di energia, ma moltiplicata per il numero di volte in cui il codice viene eseguito su milioni di dispositivi e data center, il consumo totale aumenta esponenzialmente.
Viene naturale chiedersi quanto tecniche, linguaggi e altre componenti astratte abbiano un peso determinante in questo contesto o se sia l'hardware ad avere l'impatto maggiore. Vale quindi la pena di impiegare sforzi ed energie verso questa direzione?
In questa ricerca troviamo dati che rispondono positivamente alla nostra domanda: la programmazione e lo sviluppo software sono componenti chiave dell'informatica e altrettanto chiave è il loro ruolo nelle emissioni energetiche, le stime dicono infatti che nel solo 2020 circa l'1,3% delle emissioni globali erano attribuibili allo sviluppo software. Ciò significa che lo sviluppo e l'esecuzione del software da soli rappresentano oltre il 60% delle emissioni informatiche globali [16] [24].

Nelle ricerche effettuate sono stati individuati linguaggi, tecniche di programmazione e strumenti di sviluppo per una programmazione più sostenibile [19], così come sono state individuate combinazioni di architetture e varie configurazioni tra linguaggi, loro implementazioni e ambienti d'esecuzione.
In [16] vengono analizzate proprio alcune di queste componenti e l'applicazione delle indicazioni fornite può portare a una riduzione fino al 40% della componente software delle emissioni informatiche.
Emergono quindi risultati positivi e concreti per il problema in questione ma anche che ci sono alcune componenti della programmazione e in generale dello sviluppo software in cui non viene tenuto in considerazione il fattore di consumo energetico o comunque ci sono grandi margini di crescita, questo suggerisce un potenziale risparmio mancato.
A dimostrazione di ciò in uno studio analizzato sull'impatto della versione del compilatore sul consumo energetico di un linguaggio di programmazione [29] risulta che nonostante ci sia una variazione dei risultati, non si osserva alcuna tendenza evidente di miglioramento, suggerendo che l'efficienza energetica non sia un fattore considerato nell'evoluzione dei compilatori.

Ci sono vari aspetti positivi nell'adozione di pratiche di Green Coding così come diversi ostacoli e problematiche da affrontare. Vediamoli prima riassunti in tabella per poi ragionarci sopra:

Tabella 1.1: Pro e Contro dell'efficienza energetica

Pro	**Contro**
Riduzione impatto ambientale	Potenziale impatto su prestazioni
Riduzione costi energetici ed economici	Necessità di nuova formazione
Miglior reputazione per le aziende	Necessità di differenti risorse
	Difficoltà di misurazione dell'impatto

Tra gli aspetti positivi si riscontra una riduzione dei costi energetici, così come dei costi economici proprio per la riduzione di energia utilizzata. Questo porta di conseguenza anche a una riduzione dell'impatto ambientale. Come precedentemente anticipato, una maggiore sostenibilità ambientale influenza positivamente la reputazione delle aziende aumentando ancor di più i vantaggi già citati. Tale correlazione viene spiegata dal fatto che la disponibilità di maggiori risorse permetta di affrontare più facilmente i problemi legati all'utilizzo di pratiche di Green Coding.

È bene infatti considerare che esistono preoccupazioni e limitazioni legate all'adozione di queste novità, come l'impatto potenziale sulle prestazioni a favore del risparmio energetico. Non è sempre possibile ridurre i consumi senza andare ad intaccare le prestazioni. Inoltre, l'introduzione di nuove pratiche e strumenti porta come conseguenza la necessità di nuova formazione e di risorse aggiuntive. A questo si aggiungono le difficoltà nel misurare e valutare accuratamente l'impatto ambientale [19].

Concludendo, risulta quindi possibile e necessario ottenere un significativo risparmio energetico agendo con pratiche di Green Coding ed è altrettanto di vitale importanza capire come implementare questi cambiamenti in modo da superare le problematiche esistenti ed i possibili svantaggi della transizione.

Quali sono dunque queste problematiche? Prima di entrare nel dettaglio delle soluzioni è bene analizzare quali siano queste difficoltà da superare.

1.1 Principali problematiche

Nonostante l'interesse crescente verso pratiche di sviluppo sostenibile, la transizione verso il green coding presenta ancora numerose criticità, sia sul piano tecnico che organizzativo. Tali problematiche vanno affrontate con approcci sistemici e valutazioni attente al contesto applicativo.

1. Compatibilità, costi e sostenibilità della transizione

Uno degli ostacoli principali riguarda la compatibilità con sistemi e applicazioni esistenti, che non sempre supportano linguaggi o strumenti orientati all'efficienza energetica. La transizione verso linguaggi green può comportare la necessità di mantenere in parallelo vecchie e nuove infrastrutture, aumentando i costi complessivi e, paradossalmente, generando ulteriori emissioni legate al processo di migrazione [16]. Inoltre, alcuni linguaggi progettati per ridurre il consumo energetico potrebbero non garantire le stesse prestazioni computazionali dei linguaggi tradizionali.

A ciò si aggiungono costi di formazione, strumenti dedicati e supporto tecnico che rendono l'investimento iniziale rilevante. In alcuni scenari, il costo della transizione può superare i benefici ambientali ed economici attesi, richiedendo quindi un'attenta valutazione caso per caso.

2. Limitazioni nei contesti a risorse ridotte

Nel contesto dell'Industrial Internet of Things (IIoT), i dispositivi sono spesso alimentati a batteria e dotati di risorse computazionali minime. Ci sono delle situazioni in cui non si possono ridurre le prestazioni in generale o anche solo di una sua componente, sia essa il tempo d'esecuzione, la quantità di memoria utilizzata o la potenza fornita.
In queste situazioni la sfida aggiuntiva che affronta il Green Coding consiste nell'impossibilità di ricorrere a trade-off e ridurre comunque il consumo energetico, senza intaccare dunque i fattori prestazionali che devono essere mantenuti [10].

3. Formazione insufficiente e scarsa maturità culturale

A oggi, nonostante i progressi tecnologici, la sostenibilità nello sviluppo software è ancora scarsamente rappresentata nei programmi educativi [3]. Ciò limita la diffusione di pratiche sostenibili tra le nuove generazioni di sviluppatori e ne ritarda l'integrazione strutturata nei processi di sviluppo aziendali.

4. Uso eccessivo di librerie e framework

L'utilizzo di librerie e framework esterni è una prassi comune nello sviluppo moderno, perché consente di velocizzare i processi e risparmiare risorse progettuali. Tuttavia, queste soluzioni possono introdurre componenti non necessarie, portando a codice inefficiente e più pesante, che consuma più memoria, CPU e banda di quanto richiesto dalla logica effettiva dell'applicazione [17].

5. Barriere all'adozione di linguaggi green

L'adozione di linguaggi di programmazione efficienti dal punto di vista energetico incontra diverse sfide strutturali: difficoltà di integrazione con i sistemi esistenti, curva di apprendimento per gli sviluppatori, possibili compromessi in termini di performance rispetto ai linguaggi tradizionali. Per superare queste barriere, è necessaria una collaborazione attiva tra sviluppatori, accademici, aziende e stakeholder tecnologici, in modo da innovare e generare soluzioni condivise che rispondano anche ai vincoli pratici del mercato [1].

Una volta analizzate le problematiche esistenti, l'altra componente da analizzare per comprendere bene il contesto di studio e poterlo affrontare, consiste nel capire quali sono le aree in cui si registrano i maggiori consumi energetici.

1.2 Quali ambiti presentano consumi maggiori?

Oltre agli aspetti legati all'ottimizzazione del software in chiave energetica, esistono contesti informatici di rilievo in cui lo sviluppo software può giocare un ruolo fondamentale nella riduzione del consumo energetico.

Un caso emblematico è quello delle criptovalute, il cui impatto ambientale è da tempo al centro di un acceso dibattito [18]. Il processo di mining, essenziale per validare le transazioni e generare nuove unità di valuta digitale, è infatti ad altissima intensità energetica, in particolare nei sistemi basati su Proof of Work (PoW). Questo meccanismo richiede l'impiego di enormi quantità di potenza computazionale per risolvere problemi matematici complessi, garantendo così la sicurezza della rete. Tuttavia, l'impatto energetico è significativo: basti pensare che il consumo annuale di energia della rete Bitcoin è paragonabile a quello di interi paesi come la Norvegia o l'Argentina, secondo il Cambridge Bitcoin Electricity Use Index [18]. In questo scenario, l'ottimizzazione degli algoritmi di mining, l'adozione di alternative più sostenibili come il Proof of Stake, o lo sviluppo di software di gestione dell'energia più efficiente, possono contribuire a ridurre l'impatto ambientale del settore.

Un'altra area critica è rappresentata dall'intelligenza artificiale (AI), in particolare nei modelli di grandi dimensioni. L'addestramento di un modello come GPT-3 di OpenAI può richiedere fino a 1.287 MWh di energia, generando un quantitativo di CO_2 equivalente a quello prodotto in un anno da 125 abitazioni statunitensi medie [26]. Anche in questo caso, lo sviluppo software assume un ruolo chiave. La fase di addestramento e quella di inferenza (cioè l'utilizzo del modello dopo l'addestramento) hanno esigenze diverse in termini di risorse, e quindi richiedono scelte differenti sia per quanto riguarda il linguaggio di programmazione sia per l'implementazione degli algoritmi [22]. Studi recenti suggeriscono che non solo l'algoritmo scelto, ma soprattutto il modo in cui viene implementato incide profondamente sull'efficienza energetica finale [22].

Infine, non si può ignorare il contributo delle infrastrutture IT come i data center, che rappresentano oggi una delle principali fonti di consumo energetico del settore. Le strategie per ridurne l'impatto includono ottimizzazioni strutturali, l'adozione di tecnologie di virtualizzazione, e l'implementazione di politiche intelligenti a livello di piattaforma [16].
A queste si aggiungono le opportunità offerte dal cloud computing, dove il consumo energetico e il risparmio di carbonio possono variare sensibilmente a seconda della regione geografica e della domanda di calcolo. Sebbene l'overhead prestazionale rimanga stabile, la flessibilità di scheduling delle attività nei data center cloud consente di massimizzare il risparmio di energia

e CO_2 attraverso una distribuzione più intelligente dei carichi di lavoro [12].

In questo contesto, lo sviluppo software orientato alla sostenibilità diventa un elemento integrato di una visione più ampia. La scrittura di codice efficiente, infatti, può migliorare le prestazioni delle macchine virtuali, ridurre i cicli di utilizzo della CPU, e permettere una gestione più razionale delle risorse computazionali.

Serve quindi un approccio globale, che combini buone pratiche di sviluppo software con interventi infrastrutturali e politiche strategiche, per ottenere una riduzione significativa e duratura del consumo energetico nell'ecosistema digitale. Lo sviluppo software non è un elemento isolato, ma un fattore abilitante che, se ben gestito, può contribuire in modo determinante alla transizione ecologica del settore IT.

1.3 Come fare ad adottarlo?

Per adottare delle soluzioni atte a risolvere i problemi posti è necessario innanzitutto capire in che modo questo debba essere fatto: bisogna individuare le fonti di consumo più critiche e su quali di esse si possono apportare soluzioni efficaci, cosa si può fare e cambiare relativamente ai linguaggi di programmazione e al loro ambiente circostante? Quali sono i costi e gli svantaggi dell'adozione di determinate soluzioni?

Ci sono dei casi in cui l'efficienza energetica può essere migliorata in modo significativo senza compromettere le prestazioni [13], mentre in altri è necessario scendere a compromessi che possono risultare estremamente positivi o inefficienti in base al contesto d'esecuzione.
Alcune pratiche di Green Coding includono la virtualizzazione dei server, la gestione dinamica dell'energia e l'adozione di algoritmi, strutture dati e strumenti atti a migliorare la gestione delle risorse o che meglio si adattano al caso d'uso, così come è possibile migliorare l'efficienza energetica intrinseca del codice scritto.
Per quanto riguarda i linguaggi di programmazione, si possono intraprendere azioni riguardo gli elementi sopracitati per rendere più efficienti i linguaggi già popolari. Ad esempio, linguaggi molto diffusi come Java e Python potrebbero consumare tra il 25% e il 57% dell'energia in meno per eseguire le stesse funzioni. Ciò significa che esiste un ampio margine di miglioramento [16]. Questa inefficienza è dovuta, oltre che a inefficienze intrinseche del linguaggio, anche e in gran parte al modo in cui tali linguaggi vengono utilizzati e come il codice viene scritto, spesso non in modo efficace o ottimizzato.

Oltre al linguaggio di programmazione ci sono una serie di strumenti strettamente correlati alla programmazione come framework, librerie software e ambienti di sviluppo, anche questi possono fare la differenza. Ne esistono di versioni a basso consumo energetico che forniscono agli sviluppatori funzioni e algoritmi pre-ottimizzati che minimizzano il consumo energetico mantenendo elevate prestazioni.
Recentemente è anche possibile utilizzare l'integrazione delle tecniche e strumenti di intelligenza artificiale (AI) e machine learning (ML) per contribuire ulteriormente alla riduzione dei consumi, aspetto che verrà approfondito verso la fine della survey.

In definitiva, l'adozione di pratiche di green coding richiede un'attenta analisi delle scelte progettuali, delle tecnologie utilizzate e del contesto operativo. Non esiste una soluzione universale, ma una molteplicità di strategie che, se integrate con criterio, possono portare a un notevole risparmio energetico senza sacrificare la qualità del software.
Partiamo quindi con l'identificare quali sono le cause di maggior consumo per un programma su cui bisogna concentrarsi e analizzare maggiormente per trovare soluzioni adeguate.

2. Cosa influenza l'efficienza energetica di un programma?

Nel contesto del Green Coding, l'efficienza energetica dei linguaggi di programmazione è un fattore cruciale per ridurre l'impatto ambientale del software. Come accennato precedentemente, il consumo energetico di un programma è direttamente legato alla relazione:

Energia (J) = Potenza (W) × Tempo di esecuzione (s) [27, 28]

Comprendere come questi due fattori interagiscono è fondamentale per ottimizzare un'applicazione dal punto di vista energetico.

Sebbene si tenda a pensare che ridurre il tempo di esecuzione equivalga sempre a ridurre il consumo energetico, la realtà è più articolata. Alcuni studi dimostrano una forte correlazione lineare tra tempo e consumo energetico: linguaggi più veloci tendono generalmente a consumare meno energia [3, 17].

Tuttavia, questa correlazione non è universale né perfettamente lineare. La potenza assorbita durante l'esecuzione può variare significativamente tra linguaggi e implementazioni, anche a parità di tempo di esecuzione. Alcuni linguaggi richiedono una potenza maggiore per eseguire operazioni più velocemente, generando in certi casi un consumo superiore rispetto a un linguaggio che impiega più tempo ma assorbe meno energia media [4, 27].

Questo porta al concetto di "intervallo ideale di potenza", in cui l'efficienza energetica è massima. È stato osservato, ad esempio, che ridurre la frequenza di clock o applicare limiti di potenza (power cap) può portare a risparmi energetici anche se i tempi di esecuzione aumentano notevolmente [3, 5].

La potenza assorbita può ovviamente variare anche per altre cause, ad esempio architetture diverse performeranno diversamente.

Ricordando quindi che il consumo energetico deriva dall'hardware che esegue le istruzioni

impartite dal software, possiamo identificare la CPU come componente primaria di consumo [23], insieme alle memorie. È logico quindi pensare che le componenti di un programma che sfruttano maggiormente queste risorse siano intrinsecamente più critiche dal punto di vista del consumo energetico [16]. Queste criticità riguardano sia il modo in cui il codice viene scritto e organizzato, sia l'interazione con componenti esterni e infrastrutture di supporto. Analizziamo quindi cosa influenza l'efficienza energetica di un codice:

1. Complessità algoritmica, quantità di dati e ottimizzazione del codice

L'efficienza energetica di un programma dipende in larga misura dalla qualità dell'algoritmo e dalla quantità di dati elaborati. La complessità computazionale è uno dei principali determinanti del consumo energetico: algoritmi con maggiore complessità temporale eseguono più operazioni, impiegano più tempo, richiedono più cicli di CPU e, di conseguenza, consumano più energia [17].

Inoltre, il volume dei dati trattati ha un impatto rilevante sull'efficienza: all'aumentare della quantità di dati, le differenze tra algoritmi e linguaggi si amplificano. Una cattiva gestione della complessità in presenza di grandi dataset può compromettere la sostenibilità energetica complessiva di un'applicazione [17].

Lo stesso discorso può essere fatto per la scelta e la gestione delle strutture dati.

2. Memoria: uso, distribuzione e RAM

La gestione della memoria è un altro aspetto fondamentale. La quantità totale di memoria usata durante il ciclo di vita di un programma è fortemente correlata al consumo energetico della RAM [28].

È importante distinguere tra uso massimo e uso totale: mentre il picco di memoria non ha impatti diretti chiari, è il carico cumulativo sulla RAM a determinare l'energia assorbita.

Più memoria viene utilizzata in modo prolungato, maggiore è l'energia richiesta, a prescindere dai picchi momentanei. Questo rende fondamentale progettare strutture dati e accessi alla memoria che minimizzino la permanenza e la quantità di dati in RAM.

A ciò si aggiungono anche la mancata deallocazione di risorse non più necessarie o l'allocazione eccessiva di memoria. Queste operazioni, specialmente in sistemi ad alta capacità o ad uso intensivo, contribuiscono sensibilmente al consumo energetico.

Oltre alla RAM, la cache CPU e in particolare la LLC (Last Level Cache), ossia l'ultima cache disponibile prima che la CPU debba accedere alla RAM, più lenta ed energeticamente costosa, incide notevolmente sul consumo. Tra i miss della cache LLC e l'aumento del consumo di

energia, è stato osservato che esiste un coefficiente di regressione pari a 0,94, il che indica una fortissima correlazione lineare tra i due fattori: circa il 94% della variazione del consumo energetico osservata nei benchmark dell'articolo in questione è dovuta al solo numero di LLC misses.
In altre parole, quando il programma accede spesso alla memoria principale (cioè ha molti LLC misses), il consumo energetico cresce quasi in modo direttamente proporzionale. [30].
Questo aumento è dovuto al fatto che ogni operazione di lettura o scrittura che non può essere soddisfatta dalla LLC sarà reindirizzata alla RAM.

3. Linguaggio, configurazione e contesto di esecuzione

Nonostante il modo in cui il codice di un programma venga scritto sia un fattore estremamente influente riguardo il consumo energetico, ci sono anche altri fattori altamente determinanti.
Uno di questi è la scelta del linguaggio di programmazione utilizzato [3, 22]. Alcuni linguaggi, per loro natura o per le librerie che utilizzano, tendono a generare codice più efficiente e meno dispendioso, mentre altri sono intrinsecamente più onerosi in termini di uso delle risorse computazionali, ad esempio C risulta il più delle volte la miglior scelta ecologica.
Anche l'implementazione stessa del linguaggio ha il suo effetto, genericamente i linguaggi compilati risultano più efficienti.
In questi casi si può tenere in considerazione anche la versione stessa del compilatore [5, 29]. Essa gioca comunque il suo ruolo ma ha un impatto minore rispetto al linguaggio scelto.
Anche l'efficienza dei runtime e delle ottimizzazioni a livello binario è altamente influente [17].
Infine si può dire che il contesto d'esecuzione è un fattore primario: determinate configurazioni di linguaggio, implementazione e runtime possono essere inefficienti in un contesto mentre possono essere una scelta ottimale in un altro.

4. Dimensione del codice

Una delle criticità più rilevanti e spesso sottovalutate è il software bloat [25], ovvero l'aumento ingiustificato della dimensione del codice dovuto all'uso eccessivo di librerie e framework esterni. Questo fenomeno si verifica, ad esempio, quando si importano pacchetti di terze parti da repository pubblici come NPM per una singola funzione, ma che trascinano con sé intere librerie con funzionalità ridondanti o non necessarie. Il risultato è un software più pesante, che richiede più memoria, aumenta il numero di operazioni computazionali e prolunga i tempi di caricamento e di esecuzione. Tutto ciò comporta un maggior assorbimento di potenza, una ridotta efficienza energetica e un'impronta ambientale più elevata. Inoltre è interessante notare che l'uso di

pacchetti non aggiornati o con vulnerabilità note può compromettere anche la sicurezza, la robustezza e la manutenibilità del software. Al contrario, un codice ben organizzato, che riduce al minimo le dipendenze, i calcoli superflui e le strutture ridondanti, consente di ottimizzare l'uso delle risorse hardware e di contenere il consumo energetico complessivo.

5. Ottimizzazione considerando l'hardware sottostante

Anche specifiche piattaforme o architetture hardware sottostanti influiscono notevolmente: la stessa applicazione può comportarsi in modo diverso se eseguita su CPU low-power, su sistemi multicore o su dispositivi embedded [16].

6. Livello di parallelismo

Un altro fattore che incide in modo significativo sull'efficienza energetica è il livello di parallelismo del programma. Secondo recenti test condotti in [30], parallelizzare aggressivamente i programmi è quasi sempre risultata essere una scelta energeticamente efficiente.
L'esecuzione parallela consente di suddividere il carico computazionale tra più core o thread, riducendo il tempo complessivo di elaborazione e permettendo al sistema di tornare più rapidamente a uno stato di inattività o basso consumo. Questo approccio sfrutta in maniera più efficace le capacità hardware disponibili e consente una maggiore efficienza nell'uso delle risorse, a patto che la struttura dell'algoritmo lo consenta.
Naturalmente, la scalabilità del parallelismo e l'overhead di sincronizzazione devono essere tenuti sotto controllo, poiché un'eccessiva complessità nella gestione dei thread o dei processi concorrenti può introdurre inefficienze. Tuttavia, se ben implementata, la parallelizzazione rappresenta un'importante strategia di Green Coding.

7. Comunicazione di rete

Anche la comunicazione di rete gioca un ruolo significativo: l'uso eccessivo della rete, ad esempio con richieste frequenti a server remoti o l'utilizzo improprio della banda, contribuisce in maniera rilevante al consumo energetico. Questo fenomeno è particolarmente accentuato nei sistemi wireless, dove ogni trasmissione comporta l'attivazione di trasmettitori radio ad alta intensità energetica.
In questo contesto si inserisce anche il problema delle REST API: il cosiddetto over-fetching (richieste che scaricano troppi dati) e under-fetching (richieste che scaricano troppi pochi dati, costringendo a fare ulteriori chiamate) sono tra le cause più comuni di inefficienza energetica nei sistemi distribuiti. Poiché le REST API forniscono dati tramite endpoint con risposte

standardizzate, spesso è difficile ottenere esattamente le informazioni necessarie in una singola chiamata. Di conseguenza, il numero e il peso delle richieste aumenta, con impatti tangibili sul consumo di energia [13].

In conclusione, per progettare software realmente sostenibile, è necessario un approccio integrato che consideri l'efficienza a tutti i livelli: dall'architettura del codice, alla scelta degli algoritmi e dei linguaggi, fino alle dipendenze esterne e all'infrastruttura di rete. In questo modo è possibile contribuire in modo concreto alla riduzione dell'impatto ambientale dell'industria del software. Le ricerche suggeriscono che, anche intervenendo in misura ridotta su questi aspetti, si potrebbe ridurre fino al 40% la componente software delle emissioni informatiche, diminuendo così l'impatto ambientale del settore.

Analizzando i precedenti punti, si possono raggruppare in 3 principali macrocategorie: implementazione e configurazione, contesto e ottimizzazioni del codice.

2.1 Implementazione e configurazione

Abbiamo parlato della scelta del linguaggio di programmazione, il compilatore, interprete o macchina virtuale utilizzati, il livello di parallelismo utilizzato, l'utilizzo della memoria e le librerie utilizzate.
Nella scrittura di un programma ci sono quindi molte componenti che vengono coinvolte. Di fronte a questa varietà, ci sono delle combinazioni che risultano più efficienti energeticamente. L'implementazione del linguaggio e del codice stesso, così come l'architettura hardware utilizzata, possono fare la differenza. L'argomento è stato anticipato in alcuni dei punti precedenti e verrà approfondito a parte successivamente.

2.2 Contesto

Nel Green Coding, non esiste una soluzione unica valida per ogni scenario. L'efficienza energetica del software è fortemente influenzata, tra le varie cose, dal contesto applicativo e operativo in cui il codice viene eseguito. Linguaggi, librerie, framework e ottimizzazioni varie devono essere scelti con consapevolezza in base alle caratteristiche dell'ambiente, alla tipologia di carico, alle funzionalità richieste e alle risorse disponibili.

Per carpire questo concetto, consideriamo alcuni esempi di contesti specifici riscontrati durante le ricerche di questa survey.
Uno di questi è il contesto del web development, dove problemi come overfetching e underfetching dei dati possono generare richieste inutili o insufficienti, causando sprechi energetici rilevanti, soprattutto su larga scala. Inoltre, quando si gestiscono grandi moli di dati o volumi elevati di richieste client, il costo si amplifica significativamente [25].

Un altro campo dove il contesto operativo è altrettanto cruciale è quello dell'intelligenza artificiale. Alcuni linguaggi risultano più efficienti durante la fase di addestramento, dove è richiesta una notevole capacità di calcolo e gestione di grandi dataset. Altri, invece, si dimostrano più performanti ed efficienti in fase di inferenza, dove l'obiettivo è ottenere risposte rapide con un impatto computazionale minimo [22].
L'uso di tecniche di Feature Selection (FS), ad esempio, può migliorare significativamente l'efficienza computazionale in applicazioni come la Network Intrusion Detection (NID), senza compromettere la precisione della classificazione. Questo approccio risulta particolarmente vantaggioso in linguaggi come Python e R, che beneficiano di un ecosistema maturo e ricco di librerie per l'AI [26].
Tuttavia, l'impatto della FS varia anche in base al linguaggio utilizzato, confermando che la combinazione tra tecniche di ottimizzazione e ambiente di sviluppo va valutata caso per caso.

Non è quindi solo la potenza computazionale a contare, ma anche l'efficienza dell'esecuzione rispetto all'obiettivo da raggiungere. Adattare il linguaggio, gli algoritmi e le tecniche di sviluppo al problema specifico è fondamentale per ottimizzare il bilancio energetico complessivo [22].

2.3 Ottimizzazioni del codice

Per ridurre l'impatto ambientale del software, è fondamentale identificare le aree principali su cui intervenire. Come già citato, anche se il consumo energetico è fisicamente generato dall'hardware, è il software a determinarne modalità e intensità di utilizzo [2]. In questo senso, parlare di "consumo energetico del software" significa fare riferimento all'energia richiesta da CPU, RAM e disco a causa delle istruzioni eseguite dal codice.

Uno degli aspetti più critici è l'allocazione della memoria, processo che coinvolge operazioni come la ricerca di blocchi disponibili, l'inizializzazione delle celle e l'aggiornamento delle strutture

di gestione [16]. Queste operazioni diventano energeticamente costose in applicazioni con grandi quantità di dati o con accessi frequenti alla memoria. Ottimizzare le strutture dati, ridurre la frammentazione e limitare le allocazioni dinamiche può comportare un significativo risparmio energetico, soprattutto nei sistemi su larga scala come i data center.

L'attività della CPU, definita come la percentuale di tempo in cui il processore è impegnato in calcoli, rappresenta un altro nodo cruciale. Linguaggi o ambienti che producono codice meno efficiente aumentano la pressione sulla CPU, provocando sprechi energetici evitabili [16]. Un linguaggio che consente di ottenere lo stesso risultato con meno cicli macchina consuma meno energia, a parità di hardware.

Un altro punto di intervento è quindi il miglioramento del codice e delle librerie. Codice ridondante, funzioni non ottimizzate o uso eccessivo di framework generalisti può aggravare l'impronta energetica del software. Viceversa, l'utilizzo di framework leggeri ed efficienti può produrre benefici misurabili in termini ambientali [23].

Anche se le differenze di consumo medio tra varie configurazioni software possono sembrare minime, il loro impatto diventa rilevante nel tempo. Uno studio effettuato in [23] ha mostrato che una REST API ben configurata, anche se serve un numero limitato di utenti, può far risparmiare fino a 40 kWh all'anno e ridurre le emissioni di CO_2 di circa 23 kg rispetto a una versione inefficiente.

3. Esistono linguaggi migliori di altri?

Tra i punti precedentemente elencati che influenzano l'efficienza energetica di un programma, i linguaggi di programmazione sono sicuramente tra i più interessanti da analizzare nello specifico: la varietà e la molteplicità di linguaggi rendono possibili numerose scelte che offrono determinati vantaggi e svantaggi, determinanti in base all'obbiettivo che ci si pone, sia esso puramente rivolto all'efficienza energetica o, eventualmente, a un trade-off tra quest'ultima e altri fattori.

L'efficienza energetica di un linguaggio non è un valore assoluto, ma una risultante di più fattori interdipendenti: velocità, potenza utilizzata, uso della memoria, struttura del codice, hardware sottostante, implementazione e contesto.

Ci si chiede quindi se esistano dei linguaggi generalmente migliori di altri.
Si nota una distinzione tra le due grandi famiglie di linguaggi: compilati (es. C, C++) e interpretati (es. Python, Perl, R). I primi, convertendo il codice in istruzioni macchina eseguibili direttamente, tendono a essere più performanti dal punto di vista energetico rispetto ai secondi, che si affidano a macchine virtuali o interpreti, i quali aggiungono dei passaggi intermedi introducendo un overhead costante [23] [25].
Tuttavia, la correlazione tra velocità e consumo energetico non è sempre diretta: linguaggi più lenti non sono necessariamente più inefficienti. Ad esempio, in dei test effettuati utilizzando il set di benchmark CLBG, il linguaggio Lua, pur essendo circa il 12% più lento di Perl, consuma il 53% in meno di energia in [4]. Similmente, Chapel impiega il 55% in meno di tempo rispetto a Pascal in alcuni task, pur consumando solo il 10% in più di energia [27] [28]. Questi dati suggeriscono che la potenza media assorbita durante l'esecuzione varia e che la relazione tra tempo e energia non è sempre lineare.
Bisogna anche sottolineare che la superiorità di un linguaggio non è assoluta, ma cambia in funzione del contesto applicativo. Per esempio, linguaggi come Go e Java, pur non raggiungendo le performance di C, mostrano una buona efficienza energetica e stabilità nei risultati.

Diversamente, C#, pur essendo simile a Java per design, ha mostrato un consumo fino al 400% superiore. Questo picco in particolare è stato misurato durante l'esecuzione di algoritmi di ordinamento con grandi volumi di dati [17].

All'interno di questo panorama emergono anche linguaggi progettati espressamente con una filosofia sostenibile, come Green: un linguaggio orientato agli oggetti che punta a ridurre attivamente l'uso di CPU e memoria. Compilato in C per mantenere compatibilità e prestazioni, Green combina innovazioni tecniche (come garbage collection leggera e riflessione dinamica) con una visione culturale volta a contrastare lo spreco energetico [11]. Un altro linguaggio che viene particolarmente incontro alle esigente del green coding è Zig, il quale enfatizza la sicurezza e la manutenibilità, con una politica di "nessuna allocazione implicita della memoria" eliminando quindi allocazioni implicite per ridurre gli sprechi [14].

In sintesi, non esiste un linguaggio universalmente migliore, ma esistono linguaggi che in media risultano generalmente più efficienti. C e Rust rappresentano oggi le scelte più consolidate per chi punta all'efficienza.
Il linguaggio C, in particolare, si distingue in maniera netta: è sistematicamente tra i più efficienti sia in termini di energia consumata, sia per quanto riguarda velocità e uso della memoria. In [16], viene riportato un test che cito per dare un metro di paragone concreto eseguito su un set di benchmark CLBG (Computer Language Benchmarks Game) la quale è una piattaforma di confronto tra linguaggi di programmazione che usa problemi computazionali ben definiti. In questo test, un programma C ha richiesto in media solo 57 joule per essere eseguito, contro i 4.604 joule di Perl, segnando un divario impressionante di oltre 80 volte. C è risultato anche il più veloce e il meno costoso in termini di memoria, davanti a Haskell e Python, con quest'ultimo che ha evidenziato prestazioni significativamente inferiori sotto tutti i profili analizzati nei benchmark utilizzati [3].

È bene ricordare che anche altri linguaggi se ben implementati, aggiornati e ottimizzati, possono offrire buone prestazioni. La sostenibilità, quindi, non è solo una proprietà del linguaggio, ma dell'intero ecosistema che lo accompagna, includendo compilatori, librerie, ambienti di esecuzione e soprattutto il modo in cui il codice viene scritto e gestito.

3.1 Implementazione di un linguaggio

Nel valutare l'efficienza energetica di un linguaggio di programmazione, è essenziale distinguere tra il linguaggio in sé e la sua implementazione concreta. Un linguaggio può presentare diverse varianti di compilatori o ambienti di esecuzione, ciascuno con caratteristiche tecniche che influenzano in modo rilevante il consumo energetico [16].
Per esempio, linguaggi molto diffusi come Python o Java offrono numerose implementazioni (CPython, PyPy, GraalVM, HotSpot, ecc.), che variano per modello di ottimizzazione, gestione della memoria e performance. È quindi incompleto determinare l'efficienza energetica di un linguaggio senza considerare il contesto implementativo specifico, così come quello d'esecuzione. Quindi diverse implementazioni di uno stesso linguaggio possono variare significativamente in termini di performance.

Un aspetto centrale riguarda il modello di esecuzione:

- Linguaggi compilati

  Linguaggi come C, C++, Fortran, Rust, Go, Pascal, essendo compilati direttamente in codice nativo, tendono a essere più efficienti in termini di tempo di esecuzione e consumo energetico rispetto a quelli che richiedono l'intermediazione di una macchina virtuale [2]. Inoltre, hanno un controllo più fine sull'allocazione delle risorse. In media, richiedono solo 14 J di energia e 125 MB di memoria per eseguire i benchmark, rendendoli i più efficienti in termini assoluti [27].

- Linguaggi basati su macchina virtuale

  Linguaggi come Java e C# invece bilanciano efficienza e portabilità. La VM introduce un piccolo overhead, ma i linguaggi basati su VM si comportano ancora discretamente, con consumi medi di 52J di energia e 285MB di memoria. Java, in particolare, è stato tra i linguaggi più stabili ed efficienti in numerosi contesti, pur essendo più verboso e pesante in fase di sviluppo [23, 5].

- Linguaggi interpretati

  I linguaggi interpretati (come Python, Ruby, R, Perl, Lua) offrono la massima flessibilità ma sono significativamente più dispendiosi sia in termini energetici (media di 236 J) sia di memoria (media di 426 MB). Ad esempio, linguaggi come JRuby, Perl e Python sono risultati tra i più lenti e i più energeticamente costosi nei benchmark su REST API e AI, alcuni arrivano a consumare oltre 100 volte più energia e tempo rispetto a linguaggi

compilati [22, 27, 5].

Suddividendo invece i linguaggi per paradigma di programmazione si evince che:

- Linguaggi imperativi

  I linguaggi imperativi (es. C, Pascal) mostrano i consumi più bassi: solo 116 MB di memoria media, grazie al loro approccio essenziale e poco astratto.

- Linguaggi object-oriented e funzionali

  I linguaggi object-oriented (es. Java, C++) e funzionali (es. Haskell, OCaml) si attestano intorno ai 250 MB.

- Linguaggi di scripting

  I linguaggi di scripting (es. Python, Perl, Lua), progettati per facilità d'uso e rapidità di sviluppo, sono anche i meno parsimoniosi: fino a 421 MB di memoria utilizzata [27].

Un fattore determinante diventa quindi anche la versione e il tipo di compilatore [29] o interprete, la cui scelta con relativi flag di ottimizzazione può incidere in modo significativo sull'efficienza energetica complessiva.
Studi recenti dimostrano che compilatori ottimizzati per l'architettura specifica del sistema target possono ridurre sensibilmente il tempo di esecuzione, e di conseguenza il consumo di energia.
Riportando alcuni dati raccolti, per quanto riguarda Python e i suoi interpreti, PyPy è circa 1,25 volte più veloce di CPython, mentre per Lua, LuaJIT supera l'interprete Lua di 5 volte in velocità, con conseguenze dirette anche sul consumo energetico [30].

Si può quindi dire che una stessa funzione può avere consumi molto diversi a seconda del compilatore, interprete o macchina virtuale utilizzato, delle opzioni attivate e dell'ambiente di esecuzione scelto. La sostenibilità del codice, quindi, non dipende solo da che linguaggio si usa, ma soprattutto da come lo si implementa e, come più volte citato, anche dal contesto d'esecuzione.

3.2 Quando certi linguaggi hanno migliori prestazioni

Lo scopo di questa sezione è quello di analizzare diverse situazioni di applicazione del Green Coding puntando l'attenzione dapprima su fattori di efficienza energetica e prestazionali e come questi si rapportino tra loro, mirando a individuare i migliori linguaggi in base a quali fattori

sono di maggiore importanza.
Una seconda fase analizza invece dei contesti specifici in cui è presente del codice e il Green Coding può agire.

3.2.1 Efficienza energetica e prestazioni

Come abbiamo visto, il tempo di esecuzione è uno dei due macro fattori che influenzano il consumo energetico, così come l'allocazione e la gestione della memoria è un'altra componente impattante.
Questi fattori concorrono entrambi a loro modo al risparmio energetico ma in alcuni casi può essere preferibile o necessario considerarne uno come primario rispetto all'efficienza energetica in una situazione di trade off, come ad esempio in dispositivi wearable con risorse limitate o sistemi real time in cui l'esattezza temporale è imprescindibile.
In alcuni linguaggi, tempo di esecuzione e consumo energetico vanno spesso a pari passo come per Swift e TypeScript [28]. Tuttavia, questa relazione non è sempre lineare. Ad esempio, Lua può consumare la metà dell'energia di JRuby pur avendo tempi di esecuzione quasi identici [4]. Allo stesso modo, C# consuma circa 4 volte più energia e impiega 9 volte più tempo di C nel caso del binary-trees benchmark.
I dati riportati provengono da test ripetuti ed estensivi in cui vengono utilizzati 10 benchmark del CLBG per analizzare 27 linguaggi diversi.

Volendo dare priorità all'allocazione della memoria come fattore cruciale, osserviamo come precedentemente detto che i linguaggi compilati tendono a essere più efficienti nell'uso della memoria, così come nel consumo energetico della RAM rispetto ai linguaggi su macchina virtuale o interpretati [27]. Ad esempio, linguaggi come Pascal, Go, C, Fortran e C++ sono tra quelli che utilizzano meno memoria, mentre JRuby, Dart, Erlang, Lua e Perl sono tra i più dispendiosi [27].
Il consumo energetico della RAM è fortemente correlato all'uso totale della memoria [28], con C, Rust, C++, Ada e Java tra i linguaggi che consumano meno energia correlata alla RAM [27].
È importante notare che il consumo energetico basato sulla CPU rappresenta sempre la maggior parte dell'energia consumata (in media circa l'89% per tutti i tipi di linguaggi), con la parte restante assegnata alla RAM [27, 28] e che il loro rapporto si mantiene abbastanza stabile nei compilati (min 85%, max 91%) e nelle VM (min 86%, max 92%), risultando più variabile negli interpretati (min 81.57%, max 92.90%) [27], questo suggerisce che ottimizzare un programma

per ridurre il consumo energetico della CPU può ridurre anche quello della RAM.

Quando si vogliono considerare congiuntamente energia e tempo, spesso è possibile identificare un linguaggio migliore che nella maggior parte dei contesti risulta essere C [27, 28]. Tuttavia, in benchmark specifici, altre soluzioni sono risultate più efficienti. Anche in questo caso emerge che i linguaggi compilati sono nettamente i più performanti [27].

Quando si considerano tutti e tre i fattori, la scelta del linguaggio ottimale può diventare più complessa. Ad esempio, Pascal è migliore per energia e memoria, ma peggiore per tempo di esecuzione rispetto ad altri linguaggi [27]. Analogamente, nel caso si considerino tempo d'esecuzione e uso della memoria, Rust pur essendo molto efficiente dal punto di vista energetico, scenderebbe di nove posizioni tra i linguaggi analizzati [27, 28]. C, che fin'ora è sempre risultato il migliore, risulta equivalente a Pascal in questo caso [28]. In queste situazioni, lo sviluppatore deve intervenire e decidere quale aspetto sia più importante per poter scegliere un linguaggio [27]. Genericamente si può affermare che le migliori scelte ricadono sui linguaggi compilati o su Java che risalta nonostante sia interpretato.

3.2.2 Contesti specifici

Ci concentriamo ora invece su alcuni contesti reali che sappiamo essere energeticamente costosi nel mondo dell'informatica.

- AI

  Nel contesto dell'AI, in generale, C++ e Java si confermano i linguaggi più efficienti ma la scelta dipende dalla fase predominante nell'applicazione.

  C++ è il più efficiente in addestramento, seguito da Java e MATLAB, mentre Java è il più efficiente in inferenza, consumando fino a 54 volte meno energia di R, 35 volte meno di Python e 2 volte meno di C++ [22, 26].

- RISORSE LIMITATE

  In contesti in cui si dispone di risorse limitate, come in sistemi embedded e IoT, troviamo genericamente ancora i linguaggi compilati tra i più efficenti.

  In particolare, linguaggi a controllo esplicito della memoria quali C, Zig e Rust [21, 28] evitano overhead di runtime e Garbage Collector, il che contribuisce a ridurre i consumi;

Zig in particolare consente allocatori espliciti (es. FixedBufferAllocator) e razionalizza il consumo tramite refactoring no-alloc.

– POWER CAP

Laddove tra le risorse limitate ci sia la potenza, per scelta o per necessità, si parla di power cap. La potenza può essere anche limitata volontariamente in situazioni in cui il tempo d'esecuzione non viene considerato importante. Questa tecnica è stata brevemente anticipata e verrà approfondita in un paragrafo a parte della survey.

In [5] si testano diversi livelli di power cap individuandone uno ideale. I risultati vengono poi messi a confronto con l'esecuzione senza power cap. In questa situazione abbiamo Julia e Dart tra i linguaggi interpretati più efficienti, C, C++ e Rust che continuano a essere i migliori compilati in termini di efficienza. Altri linguaggi con risultati degni di nota sono Haskell che mostra la maggiore riduzione energetica (-27,3%) e una delle minori riduzioni di prestazioni (+84,35%) rendendolo in questa classifica il più efficiente in assoluto. In contrasto Julia mostra il risparmio minore (-6,3%) a fronte comunque del minor rallentamento nei tempi d'esecuzione (+35,34%). Java migliora dal 13% al 17%, ma raddoppia i tempi di esecuzione, non sembrando quindi un buon candidato per il power capping. Infine Ruby e Perl risultano i peggiori consumando oltre 100 volte più energia e tempo rispetto a C e C++.

- SVILUPPO WEB

Come vedremo nel dettaglio nel capitolo 4, in un contesto di utilizzo di API REST, Python risulta, tra i linguaggi analizzati, il più parsimonioso in termini di consumo energetico, se si vuole però considerare l'efficienza energetica il miglior candidato risulta essere Javascript/Typescript.

- DATABASE

Come vedremo nel dettaglio nel capitolo 5, l'utilizzo di NoSQL può aumentare di molto l'efficienza energetica, bisogna però tenere in considerazione che se dal lato client non si adoperano le dovute accortezze e ottimizzazioni, l'efficienza complessiva del sistema può peggiorare.

Come può essere intuibile, in generale valgono le stesse indicazioni date per le situazioni di risorse limitate: linguaggi con un controllo maggiore della gestione della memoria portano ad eseguire meno operazioni di lettura e scrittura riducendo di conseguenza il consumo energetico.

3.3 I linguaggi Green e Zig

Come accennato esistono anche linguaggi progettati nativamente per essere sostenibili. Tra questi Green e Zig che andiamo ora ad analizzare.

3.3.1 Green

Green è un linguaggio creato da José de Oliveira Guimarães [11] focalizzato su efficienza e semplicità, progettato per aiutare gli sviluppatori a scrivere codice pulito e performante con un runtime minimale.
Il linguaggio Green rappresenta un caso apparentemente unico nel panorama dei linguaggi orientati agli oggetti, unendo static typing, riflessione e meta-programmazione avanzata. Le scelte progettuali e la gestione efficiente delle risorse rendono questo linguaggio un esempio di come l'architettura di un linguaggio possa, indirettamente, favorire pratiche di sviluppo software più sostenibili.
Green presenta caratteristiche che, lette in chiave Green Coding, possono contribuire alla sostenibilità, quali:

- Efficienza nella gestione delle risorse

  Il garbage collection e la tipizzazione statica aiutano a prevenire memory leak e riducono il consumo di risorse computazionali, con potenziali benefici energetici.

```
1    var any : AnyClassObject = integer;
2    var anArray : AnyArray;
3
4    // creates a 10-element integer array
5    anArray = array(any[].new(10);
6    anArray.set(5, 0); // anArray[0] = 5;
7    anArray.set(3, 1); // anArray[1] = 3;
8
9    Out.writeln( anArray.get(0) ); // print 5
```

Codice 3.1: Garbage collection e tipizzazione statica: uso di array creati a runtime e tipati senza gestione manuale della memoria

- Modularità e riusabilità

  La separazione tra ereditarietà e subtyping, unita alle classi parametrizzate, incoraggia la scrittura di codice riutilizzabile e meno ridondante, riducendo il tempo di compilazione ed esecuzione e, quindi, l'energia necessaria.

```
class List(T)
public:
...
    proc write()
    begin
        var i : integer;
        i = 0;
        while v[i] <> GreenCompiler.getEofOfList(T) do
        begin
            Out.writeln( v[i] );
            ++i;
        end
    end
    ...
private:
    var v : array(T)[];
end // List(T)
```

Codice 3.2: Subtype/Sottoclasse e Classi parametrizzate: implementazione unica per tutti i tipi T: meno codice e tempi di build

- Meta-programmazione mirata

  L'uso di shells e riflessione tipata consente di adattare il comportamento del software senza ricompilazioni complete, riducendo il ciclo di sviluppo e test e limitando l'impatto computazionale [6].

```
class Counter
proc init()
begin
    n = 0;
end
public:
    proc add( s : integer ) begin n = n + s; end
    proc get() : integer begin return n; end
private:
    var n : integer;
end
... // shell che incrementa e delega a super.draw()

var c : Counter;
var w : Window;

w = Window.new(...);
c = Counter.new();

try(catch)
    Meta.attachShell( w, DrawCount.new(c) );
end
...
Out.writeln("draw was called ", c.get(), " times");
```

Codice 3.3: Shells e Riflessione tipata: approccio dinamico che consente adattamenti selettivi senza rebuild globale

- Astrazione ad alto livello

  Il trattamento delle classi come oggetti di prima classe permette di semplificare operazioni complesse, diminuendo il codice boilerplate e migliorando l'efficienza complessiva.

```
var ci : ClassInfo;
var p : Person;
...
ci = p.getClassInfo();

var v : array(ClassMethodInfo)[];
v = ci.getMethods();
for i = 1 to v.getSize() - 1 do
    Out.writeln( v[i].getName() );

var objInfo : AnyObjectInfo;
objInfo = p.getInfo();

// list names of all instance variables
var f : array(ObjectInstanceVariableInfo)[];
f = objInfo.getInstanceVariables();
for i = 0 to f.getSize() - 1 do
    Out.writeln( f[i].getName() );
```

Codice 3.4: First Class e Riflessione Introspettiva Tipata: semplifica operazioni complesse riducendo boilerplate e si ispeziona/invoca senza generare/duplicare codice ad hoc

Purtroppo non si trovano altri articoli dedicati specificamente al linguaggio Green successivi al paper del 2013. Questo indica che potrebbe non esserci stata un'ampia diffusione nell'ambito accademico o nella comunità online per questo linguaggio. È stato identificato un linguaggio sviluppato sempre da Guimarães chiamato Cyan che riprende delle meccaniche di Green (shells / metaoggetti dinamici, riflessione/metaprogrammazione, tipizzazione statica, sistema di eccezioni "object-oriented") [7], questo però non è orientato alla sostenibilità per cui non verrà trattato in questa survey.

3.3.2 Zig

Un altro linguaggio con una vocazione alla sostenibilità ambientale è Zig, linguaggio di programmazione low-level orientato a prestazioni e prevedibilità, ossia alla riduzione al minimo di comportamenti "nascosti" o non deterministici del codice, così che il programmatore possa sapere in anticipo cosa farà il programma e quanto costerà in termini di risorse (tempo, memoria, energia). Zig adotta infatti come principio fondamentale il "no implicit memory allocation", eliminando così la necessità di garbage collector e favorendo una gestione esplicita della memoria da parte dello sviluppatore.

Questa caratteristica, unita alla disponibilità di allocatori selezionabili e ottimizzabili (come Page Allocator o Fixed Buffer Allocator), consente di ridurre al minimo le allocazioni non necessarie, con impatti diretti sul consumo energetico.

Grazie al controllo fine delle risorse, all'assenza di overhead tipici di runtime e garbage collector e a un compilatore altamente ottimizzato, Zig si propone come un'opzione particolarmente adatta non solo per lo sviluppo di applicazioni di sistema, ma anche per scenari embedded e software ad alta efficienza, dove ogni ciclo di CPU e ogni Joule risparmiato hanno un peso rilevante.

Zig è quindi pensato per spostare lavoro dal runtime al compile-time, ridurre allocazioni e garbage collector e rendere esplicite le risorse.

Vengono di seguito forniti alcuni snippet di codice reperiti da vari articoli, i quali evidenziano alcune caratteristiche distintive di Zig in chiave green.

```
// Zig rifiuta conversioni implicite int->ptr e deref di puntatori non-nullable = 0.

pub fn main() void {
    // var p: *i32 = 0; // NO, non compila (conversione implicita vietata)
    // _ = p.*; // NO

    // Anche la conversione esplicita richiede un puntatore *nullable*:
    var q: ?*i32 = @intToPtr(?*i32, 0); // ok: può essere null
    if (q) |ptr| {
        _ = ptr.*; // (non verrà eseguito: q ènull)
    }
}
```

Codice 3.5: Type System: errori di memoria scoperti in compilazione (o con i controlli runtime in debug) evitano crash, riavvii e cicli di diagnosi, meno CPU sprecata e meno lavoro ripetuto. [15]

```
    // @cImport(@cInclude("stdio.h")) importa simboli C e li chiami direttamente.
    const stdio = @cImport(@cInclude("stdio.h"));

    pub fn main() void {
        _ = stdio.puts("hello world"); // chiama direttamente puts() della libc
}
```

Codice 3.6: Interoperabilità C senza glue code: riuso di librerie C iper-ottimizzate senza generare glue code o bindings, quindi meno codice, meno allocazioni/accessi superflui, quindi meno tempo di impiego della CPU. [15]

```
    const std = @import("std");

    pub fn main() !void {
        var buf: [10]u8 = undefined; // buffer piccolo, locale (stack)
        for (0..buf.len) |i| buf[i] = 0;

        // std.heap.FixedBufferAllocator per allocare dentro un buffer fisso (anche sullo stack)
        var fba = std.heap.FixedBufferAllocator.init(&buf);
        const alloc = fba.allocator();

        const tmp = try alloc.alloc(u8, 5); // alloca dentro buf
        defer alloc.free(tmp); // rilascio O(1), nessuna syscall

        // ... usa tmp ...
    }
```

Codice 3.7: Allocatori espliciti: niente chiamate al sistema/heap per oggetti piccoli e temporanei, zero frammentazione, prevedibilità di cache. [9]

```
const std = @import("std");

// Tagged union (il tag èl'enum generato da 'union(enum)').
const Registry = union(enum) {
    core: u32,
    plugin: []const u8,
    builtin: void,
};

fn describe(r: Registry) []const u8 {
    return switch (r) {
        .core => "core registry",
        .plugin => "plugin registry",
        .builtin => "builtin registry",
    };
}

test "tagged union + switch esaustivo" {
    try std.testing.expectEqualStrings("core registry",
        describe(.{ .core = 42 }));
}
```

Codice 3.8: Tagged union e switch esaustivo: rami mancanti (bug) vengono scoperti in build e meno error handling dinamico, dunque meno rami imprevedibili e meno spreco della CPU. [9]

Per concludere, entrambi i linguaggi rappresentano un cambio di paradigma, in cui la sostenibilità non è un'aggiunta, ma una proprietà fondamentale del linguaggio.

4. Configurazioni e implementazioni più sostenibili

Avendo analizzato fin'ora nel dettaglio gli aspetti più strutturali, vediamo come questi possono combinarsi e quali risultati ottengono.

Esponiamo le varie scelte che si hanno per comporre la propria configurazione, con un approccio bottom up di un sistema.
Anche se non è al centro delle nostre analisi partiamo dall'hardware dicendo che anch'esso incide in modo sostanziale sulle prestazioni energetiche. Viene riportato un esempio a riguardo in cui vengono confrontati un MacBook Air Apple Silicon e un desktop AMD. Il primo risulta molto più efficiente, impiegando solo il 25% del tempo e il 10% dell'energia per lo stesso compito, con un incremento di consumo energetico molto più contenuto all'aumentare del carico. [14].

Guardando invece all'implementazione abbiamo ripetuto più volte la superiorità in termini di efficienza energetica dei linguaggi compilati, vale però la pena notare che ci sono eccezioni quali Java e C# i quali ottengono comunque generalmente ottimi risultati.
In base a ciò, si pongono quindi diverse scelte riguardo anche alla versione del compilatore, macchina virtuale o interprete che, come dimostrato dai vari studi analizzati, hanno un impatto diretto sul consumo energetico e sulle prestazioni. Riportiamo quindi a riguardo alcune conclusioni dei suddetti studi.
Riguardo al compilatore, per C la versione più efficiente è Gcc Ubuntu 9.4.0-1ubuntu1 20.04.2, mentre versioni precedenti o successive consumano di più sia in termini di tempo che di energia [29].
Per quanto riguarda Java abbiamo che OpenJDK 11.0.20.1 e 16.0.1 risultano le versioni più ecologiche [29].
In Python invece, la versione migliore del suo interprete risulta essere la 3.7.16 che offre un

bilanciamento ottimale tra consumi e tempo.

Oltre a tutto ciò, il codice prodotto dai linguaggi di programmazione viene influenzato anche da elementi di runtime quali l'ambiente d'esecuzione e i framework utilizzati.
Come già detto, il contesto d'esecuzione è spesso determinante quando si parla di efficienza e consumo energetico.
In [23] vengono valutati, nel contesto di utilizzo di REST API, 5 framework e 5 ambienti d'esecuzione: Spring Boot e Micronaut (Java), Express e Nest (JavaScript / TypeScript) e Django (Python) per i framework e OpenJDK e GraalVM (Java), Node.js e Bun (JavaScript / TypeScript) e CPython e PyPy (Python) per gli ambienti.
Da questa analisi emerge che Python, in combinazione con Django e PyPy, risulta il migliore in quanto a consumo energetico utilizzando sia meno CPU che meno memoria, entrambi fattori determinanti.
Per quanto riguarda però l'efficienza energetica spicca invece Javascript/Typescript con la combinazione tra Express e Bun che rivela buoni risultati anche per quanto riguarda il consumo energetico, rendendolo quindi la scelta migliore. Questo perchè a parità di lavoro svolto risulta più efficiente.
Nel particolare, considerando come fattore primario l'efficienza energetica si ha che:

- Per Javascript/Typescript risulta migliore Express + Node
- Per Python risulta migliore Django + PyPy
- Per Java risulta migliore Spring + OpenJDK

Se invece alcune risorse si mostrano limitate, abbiamo anche una vista su quali combinazioni risultino più efficienti, ricordando che comunque una migliore gestione di questi parametri ha un impatto positivo sull'efficienza energetica:

- Per minor uso di memoria:
  - Python risulta il migliore tra tutti in particolare con la combinazione Django + CPython
  - Per Java abbiamo Micronaut + OpenJDK
  - Per Javascript/Typescript abbiamo Nest + Node

- Per minor uso di CPU:
  - JavaScript/TypeScript risulta il migliore tra tutti in particolare con la combinazione Nest + Node
  - Per Java abbiamo nuovamente Micronaut + OpenJDK
  - Per Python abbiamo ancora Django + CPython

Notiamo infine che ci sono altre scelte apparentemente minori per quanto riguarda la configurazione di runtime come la scelta dell'allocatore di memoria, il quale può generare miglioramenti nell'efficienza energetica. In [20] viene testata su CPython la sostituzione dell'allocatore di memoria di default con altri esistenti quali Glibc malloc, Jemalloc e Mimalloc, tra questi Jemalloc migliora prestazioni e consumo energetico in alcuni casi anche oltre 1,6 volte.

5. Algoritmi, strutture dati e tecniche di programmazione in rilievo

Nel contesto del Green Coding ci sono altre scelte implementative che influenzano la sostenibilità energetica oltre a quelle di runtime. Queste altre scelte sono quelle di progettazione e implementazione del codice.
Di queste, abbiamo prima parlato della scelta dei linguaggi di programmazione che verrà ripresa anche in questo capitolo, in relazione anche alle scelte degli algorimi e delle strutture dati utilizzate, nonchè varie tecniche di programmazione che riportano risultati significativi.

Come già spiegato, il contesto d'esecuzione è determinante e di conseguenza le soluzioni variano al variare di questo.
Per quanto riguarda il contesto della AI, per cui bisogna distinguere tra fase di training e fase di inferenza, sono stati raccolti dei dati provenienti da [22] in cui vengono esaminati vari linguaggi di programmazione in combinazione con vari algoritmi atti ad ottenere lo stesso risultato.
I risultati indicano un impatto significativo sulla scelta di queste componenti implementative: per la fase di training la migliore combinazione risulta essere Java con gli alberi decisionali mentre per la fase di inferenza risulta essere C++ con la regressione logistica.

In particolare, per i linguaggi di programmazione analizzati risulta che

- Per C++ e Python l'algoritmo più efficiente è la regressione logistica
- Per Java gli algoritmi più efficienti risultano gli alberi decisionali e Naive Bayes
- Per Matlab l'algoritmo più efficiente risulta quello degli aberi decisionali
- Per R l'algoritmo più efficiente risulta KNN

Nella misurazione dei consumi effettuati con CodeCarbon, in R l'implementazione dell'algoritmo della regressione logistica rappresenta oltre il 71% del consumo energetico totale sommando l'esecuzione di tutti gli algoritmi analizzati, risultando quindi il più costoso.

In Python, l'algoritmo SVC (Support Vector Classifier) incide allo stesso modo per oltre il 63% mentre in Java, l'albero decisionale può arrivare fino al 52%.

Uno studio [28] condotto su un totale di 27 linguaggi di programmazione analizza la loro efficienza energetica, tra i vari modi, attraverso l'esecuzione di algoritmi di ordinamento quali Merge Sort, Quick Sort e Selection Sort. Si nota subito che Selection Sort non sia un'opzione ottimale per via della sua scarsa scalabilità. Per quanto riguarda invece Merge Sort, i linguaggi in cui consuma meno sono C, Rust e Go mentre per Quick Sort sono Pascal, C e Rust.
In un altro studio [17] vengono analizzati i linguaggi Java, Go e C#. Generalmente Java risulta più performante con tutti gli algoritmi analizzati mentre in generale, l'algoritmo più efficiente risulta sempre essere Merge Sort, il quale però utilizza più memoria portando quindi la scelta ideale ad essere Selection Sort per piccoli input mentre Quick Sort per input medio-grandi.
I due studi vanno in contraddizione per quanto riguarda Java e questo è spiegato dall'implementazione utilizzata: in [28] viene notato che, utilizzando delle strutture dati List per Java, la sua esecuzione risulta inefficiente allo scalare dell'input.
Questa differenza si viene a creare poichè le List, e in generale le Collections, richiedono la creazione di nuove strutture da popolare dinamicamente, aggiungendo un overhead all'esecuzione. Questo impatta negativamente rispetto a implementazioni più dirette che operano su array preallocati [28].

Questi esempi evidenziano come la scelta dell'algoritmo e la sua specifica implementazione (inclusi dettagli come librerie, versioni e strutture dati) influenzino in modo diretto i consumi complessivi del sistema.

Passando quindi ad alcuni dei contesti in cui si riscontrano consumi importanti analizzati in questa survey, per quanto riguarda le basi di dati, notiamo che le strutture come gli indici e le covered queries in MongoDB possono migliorare le prestazioni anche di oltre 200 volte, riducendo il consumo energetico fino al 45%, così come anche l'ordinamento delle query contribuisce in modo sostanziale a ridurre il tempo di esecuzione e i consumi [13]. Questo perchè vengono evitati fetch dei documenti con conseguenti tagli drastici alle letture da disco e RAM, ci sono meno byte trasferiti e quindi meno tempo d'esecuzione della CPU, minor utilizzo di rete e meno RAM attiva.
Nel contesto cloud, algoritmi avanzati di scheduling e provisioning dei task sono fondamentali per ridurre l'uso inefficiente delle risorse. Si tratta di algoritmi che decidono in quali slot

temporali, su quali istanze cloud e in che ordine eseguire i task per minimizzare latenza e costo e massimizzare l'utilizzo delle risorse nel cloud. Gli studi analizzati [8] [12] hanno mostrato che gli algoritmi greedy permettono di sfruttare i periodi di bassa domanda per eseguire job complessi, riducendo il costo energetico e massimizzando l'utilizzo dei server.
Inoltre l'uso esclusivo di istanze riservate, molto utilizzate per ridurre i costi economici, compromette invece la sostenibilità dei cloud. Al contrario, i cluster dinamici di istanze cloud bilanciano meglio costi e sostenibilità.
Questo perchè renderebbe possibile la scelta di uno slot temporale con minore impatto energetico. Quindi un algoritmo di scheduling Carbon Time [12], che ottimizza il rapporto CO_2 risparmiata e diminuzione di prestazioni e cluster dimanici risultano l'opzione migliore per ottenere una significativa efficienza energetica.

Rimanendo sempre nell'ambito web parliamo invece delle scelte a livello di database:
NoSQL tende a spostare l'onere computazionale (join, filtri) sul client. Questo riduce i consumi lato server ma può aumentare il carico energetico sui dispositivi degli utenti, talvolta meno efficienti [25].
È fondamentale implementare buone pratiche lato client per ottimizzare la gestione dei risultati in presenza di architetture NoSQL.

Infine approdiamo al campo del machine learning in cui l'uso di tecniche di Feature Selection (FS) rappresenta una delle strategie più efficaci per ridurre i consumi computazionali mantenendo intatta la qualità dei modelli. Le tecniche FS si dividono in tre categorie: Filter, Wrapper ed Embedded.
Queste tecniche possono ridurre il numero di feature fino a un terzo, migliorando i tempi di addestramento e inferenza, con impatti minimi o nulli sulle metriche di performance finale. In ambito IDS (Intrusion Detection Systems), ad esempio, l'uso combinato di metodi come Chi-Square, Information Gain e Recursive Feature Elimination ha prodotto modelli più rapidi e meno costosi energeticamente [26].

6. Quali strumenti consentono uno sviluppo più green?

Dopo tutte le analisi finora eseguite è quindi chiaro che lo sviluppo software sostenibile è molto complesso e si articola in moltissime sfaccettature, da qui, risulta quindi la volontà di individuare degli strumenti a supporto. In questo contesto, è utile distinguere tra:

- Strumenti integrabili o abilitanti, che possono essere utilizzati direttamente per ottimizzare l'efficienza
- Strumenti di controllo e monitoraggio, per valutare e misurare i consumi energetici

Strumenti integrabili e tecnologie abilitanti

Anche librerie e framework progettati per il risparmio energetico contribuiscono allo sviluppo green. Il Green Computing Framework (GCF), ad esempio, è una libreria open-source che offre funzioni pre-ottimizzate per minimizzare i consumi. [24].

Nel mondo cloud, strumenti come GAIA, uno scheduler per job batch carbon-aware, introducono logiche di carbon-aware scheduling. GAIA consente di scegliere tra diverse strategie di allocazione delle risorse cloud e, secondo recenti ricerche, è in grado di raddoppiare il risparmio di emissioni per ogni aumento percentuale dei costi, riducendo del 26% l'overhead prestazionale [12].

Altri esempi di scheduler efficienti sono AWS ParallelCluster e Kubernetes, utilizzati per gestire carichi di lavoro dinamici. Questi strumenti sfruttano l'elasticità delle risorse cloud, adattando il numero di server attivi alle necessità, con un impatto diretto sia sui tempi di attesa che sul consumo energetico complessivo [12].

Anche strumenti nati per altri scopi si rivelano utili dal punto di vista green. È il caso di GraphQL, un linguaggio di query per API che consente di recuperare solo i dati effettivamente necessari, evitando fenomeni di over-fetching e under-fetching comuni nelle API REST [13].

Notiamo infine uno strumento utile per un campo molto specifico, ossia quello delle criptovalute che, come inizialmente spiegato in questa survey, risultano in un consumo notevole nel settore IT: nel campo della blockchain, le stablecoin rappresentano un'innovazione con implicazioni

anche ambientali. Utilizzano meccanismi di consenso meno energeticamente costosi, come Proof of Stake (PoS) o delegated PoS, che evitano il mining intensivo di calcolo [18].
Altre blockchain individuate che utilizzano questi meccanismi sono Tron, Avalanche, Algorand e Solana.

Strumenti di controllo e monitoraggio

Oltre all'uso di strumenti green-oriented, è fondamentale poter misurare e validare l'efficienza energetica delle applicazioni.
Tra i tool più noti, PowerTOP di Intel consente di identificare in tempo reale le componenti che causano consumo energetico anomalo a livello di sistema. È particolarmente utile per ottimizzare laptop o ambienti Linux [1]. Similmente, CodeCarbon [22] è un software con il compito di calcolare la quantità di anidride carbonica generata dalle risorse di computing per l'esecuzione del codice.
Altro tool particolarmente utile risulta PowerAPI, un framework middleware che consente di monitorare e ottimizzare in tempo reale l'energia utilizzata dalle applicazioni [24].
L'ETAF (Energy-aware Test Automation Framework), invece, permette di integrare la valutazione dell'efficienza energetica direttamente nelle pipeline di test automatici, aiutando gli sviluppatori a verificare l'impatto energetico delle modifiche introdotte nel codice [1].
Per il web, PageSpeed Insights di Google rappresenta uno strumento accessibile che fornisce suggerimenti specifici per migliorare le prestazioni del sito, riducendo tempi di caricamento, codice ridondante, strutture inefficienti e, di conseguenza, consumi energetici [13].
Infine, l'uso di algoritmi di AI e tecniche di Machine Learning consentono oggi di realizzare sistemi in grado di adattarsi dinamicamente al carico di lavoro, prevedere l'uso delle risorse e ottimizzare automaticamente l'allocazione. [13]. La AI e il Machine Learning, da settori di alto consumo, possono quindi diventare anche strumenti per conseguire una maggiore sostenibilità.

7. Cosa si può dire del contesto più ampio? Ossia del Green Computing

7.1 Applicazioni indirette

Sebbene il Green Coding si concentri prevalentemente sulle pratiche legate alla scrittura e ottimizzazione del codice, esistono numerose applicazioni indirette che influiscono significativamente sull'efficienza energetica complessiva. Queste pratiche appartengono maggiormente al Green Computing poichè non rientrano direttamente nel dominio dello sviluppatore, ma ne condizionano comunque gli effetti.

A livello hardware, le tecniche di gestione dell'energia come il Dynamic Voltage and Frequency Scaling (DVFS) e il power gating permettono ai sistemi di regolare tensione e frequenza in funzione del carico computazionale o di disattivare componenti inattivi, con risparmi sensibili in fase di idle o basso carico [24]. È bene notare che sebbene siano integrati nell'hardware, possono essere attivati o sfruttati tramite il software, ad esempio attraverso la pianificazione intelligente dei task o l'ottimizzazione dei thread e dei carichi di lavoro. La loro efficacia dipende da una progettazione consapevole dell'utilizzo delle risorse, capace di creare una sinergia tra codice e piattaforma sottostante [24].
Architetture ibride come big.LITTLE di ARM consentono il bilanciamento dinamico tra core ad alte prestazioni e core a basso consumo, secondo le esigenze del carico di lavoro.

In ambito infrastrutturale, il concetto di power capping consente di fissare limiti energetici a livello di CPU. Sebbene ciò comporti un peggioramento delle performance di esecuzione (con incrementi fino al 128%), i risparmi energetici medi si attestano intorno al 13,75% [5].

In scenari cloud l'edge computing si afferma come una soluzione in grado di ridurre la latenza e il consumo energetico elaborando i dati più vicino alla loro origine [1].

Nei data center distribuiti, le strategie di allocazione spaziale e temporale del carico di

lavoro stanno guadagnando rilevanza. Soluzioni come la BB-RR (Backup Battery – Renewable Regulation) abbinano l'uso intelligente di batterie e fonti rinnovabili alla geodistribuzione delle risorse computazionali, ottenendo una riduzione dei costi operativi fino al 10,88% e un incremento dell'efficienza ambientale senza penalizzare le performance [31].
Tornano qui nuovamente in gioco le tecniche di ML e AI che con la loro intelligenza adattiva sono particolarmente utili ottimizzando il raffreddamento e la distribuzione dei job, riducendo gli sprechi energetici in tempo reale [13].

Infine, lo scheduling carbon-aware rappresenta un'ulteriore strategia che sfrutta la flessibilità temporale dei job batch (non interattivi) per ritardarne l'esecuzione nei momenti di minore intensità di CO_2 della rete elettrica.
Poiché l'intensità di carbonio (gCO_2/kWh) varia nell'arco della giornata e della settimana, lo scheduler guarda alle previsioni di intensità e sceglie uno slot o una finestra di tempo a bassa CO_2 entro il margine di attesa consentito dal job, ossia la sua flessibilità temporale.
In tali casi, si osservano compromessi tra tempi di esecuzione e riduzione delle emissioni, gestibili mediante politiche intelligenti come Wait Awhile ed Ecovisor [12].

8. Si possono definire delle linee guida dai dati attualmente esistenti?

L'efficienza energetica di un sistema software dipende quindi dal coordinamento di molteplici fattori: linguaggio, algoritmo, hardware, runtime, workload, distribuzione, durata del job, e persino geografia e fascia oraria. Risulta ovvio quindi che definire delle metodologie da seguire sia alquanto intricato ma è comunque possibile tracciare una serie di raccomandazioni concrete e contestualizzate:

- Scegliere linguaggi e tecniche in funzione del contesto
  Nessun linguaggio è sistematicamente il più efficiente: l'efficienza dipende dal contesto d'uso e molti aspetti tecnici sottostanti [28].

- Ottimizzare l'algoritmo prima del codice
  Un algoritmo ben progettato riduce operazioni, uso di memoria e cicli CPU [24].
  Azioni quali ottimizzazione di loop e di gestione della memoria così come la località dei dati migliorano prestazioni e consumo [29].
  Si invita inolte a evitare calcoli ridondanti, eliminare dead code, migliorare la località della memoria e usare strutture dati contigue [21].

- Sfruttare le architetture a basso impatto
  L'edge computing consente di ridurre la latenza e risparmiare banda/energia nelle architetture distribuite [10].
  I modelli serverless spostano la complessità della sostenibilità sull'infrastruttura cloud [8].

- Valutare l'effetto del power capping
  Il power cap può ridurre il consumo energetico del 13,75%, ma spesso aumenta il tempo di esecuzione in media fino al 91% [5]. Bisogna quindi valutare in quali contesti può essere applicato.

- Applicare strategie di scheduling consapevoli

  Nei sistemi cloud, pianificare l'esecuzione in base alla durata dei job e intensità di carbonio locale consente notevoli risparmi [12].

  I job di media durata (3–12h) offrono il miglior compromesso tra flessibilità e risparmio energetico.

- Ridurre il software bloat

  Il codice superfluo aumenta l'energia usata. Minimizzare le dipendenze utilizzate richiede sicuramente una conoscienza adeguata dell'ambiente tecnico attualmente esistente [25].

- Bilanciare consumo, prestazioni e memoria

  Non esiste sempre una soluzione che ottimizza tutto: occorre bilanciare tempo, energia e memoria. In certi contesti è più opportuno scegliere strategie lente ma energeticamente più efficienti [23].

Dalle analisi complessive della survey viene proposta una scala relativa a quanto i vari aspetti dello sviluppo software impattano sulla sostenibilità ambientale:

Scelte algoritmiche e strutture dati Influiscono direttamente sull'efficienza computazionale	1
Linguaggio di programmazione e modello di runtime Alcuni linguaggi sono più efficienti di altri	2
Implementazione e librerie Le pratiche di codifica influenzano l'efficienza energetica	3
Ambiente d'esecuzione Le scelte di framework influenzano il consumo energetico	4
Versione e flag del compilatore o interprete Le ottimizzazioni del compilatore variano nel tempo	5

1. Scelte algoritmiche e strutture dati

   Incidono direttamente su tempo di esecuzione, numero di operazioni, uso di memoria, accessi I/O e cicli CPU. Ad esempio se Java usa List dinamiche in quicksort, la sua sostenibilità peggiora rispetto all'utilizzo di array statici.

2. Linguaggio di programmazione e modello di runtime (compilato / VM / interpretato)
   Alcuni linguaggi risultano generalmente più efficienti di altri. Inoltre, diversi linguaggi possono avere prestazioni differenti in contesti e ambienti d'esecuzione diversi.

3. Implementazione e librerie
   Un linguaggio può essere green o meno a seconda di come viene implementato. Lo stesso algoritmo può essere energeticamente più o meno efficiente a seconda delle pratiche di codifica e librerie. Infine, l'utilizzo indiscriminato di quest'ultime può portare ad aumenti considerevoli della dimensione del codice con conseguente spreco energetico.

4. Ambiente d'esecuzione
   Le ottimizzazioni apportabili fin'ora indicate variano drasticamente in base al contesto e all'implementazione. In generale è possibile dire che la scelta di framework, ambienti di sviluppo, tool utilizzati e come già detto librerie, portano a variazioni discrete del consumo energetico.

5. Versione e flag del compilatore o interprete
   Le varie versioni presentano ottimizzazioni del compilatore, JIT, garbage collection, gestione della memoria con un impatto ambientale variabile nel corso del tempo, l'ultima versione non è necessariamente la più efficiente.

In definitiva, la sostenibilità del software è un obiettivo multidimensionale, che si raggiunge combinando ottimizzazione algoritmica, consapevolezza del contesto esecutivo, progettazione modulare e gestione oculata delle risorse. Non si tratta solo di scrivere codice, ma di pensare sistemicamente a ogni livello del ciclo di vita del software, agendo con consapevolezza progettuale per un impatto ambientale minore.

Bibliografia

[1] Akoh Atadoga, Uchenna Joseph Umoga, Oluwaseun Augustine Lottu, and Enoch Oluwademilade Sodiya. Tools, techniques, and trends in sustainable software engineering: A critical review of current practices and future directions. *WJAETS World Journal of Advanced Engineering Technology and Sciences*, 2024.

[2] Ismael Camargo-Henríquez, Anthony Martínez-Rojas, and Gema Castillo-Sánchez. Energy optimization in software: A comparative analysis of programming languages and code-execution strategies. *IEEE Xplore*, 2019.

[3] Vasco Vieira Costa. Green computing - energy consumption for programming languages in inefficient fibonacci. *Research Gate*, 2025.

[4] Marco Couto, Rui Pereira, Francisco Ribeiroa, Rui Rua, and João Saraiva. Towards a green ranking for programming languages. *ACM Digital Library*, 2017.

[5] Simão Cunha, Luís Silva, João Saraiva, and João Paulo Fernandes. Trading runtime for energy efficiency. leveraging power caps to save energy across programming languages. *ACM Digital Library*, 2024.

[6] José de Oliveira Guimarães. Reflection for statically typed languages. 1998.

[7] José de Oliveira Guimarães. The cyan language. *ArXiv*, 2025.

[8] Sreenath Devineni, Bhargavi Gorantla, Intiaz Shaik, and Anand Kumar V. Optimizing cloud resources using algorithmic approaches in serverless computing. *IEEE Xplore*, 2024.

[9] Pedro Duarte Faria. *Introduction to Zig*. Independently published, 2024.

[10] Xavier Fernando and George Lăzăroiu. Energy-efficient industrial internet of things in green 6g networks. *MDPI*, 2024.

[11] J. Guimarães. The green language. *Elsevier, Computer Languages, Systems Structures*, 2006.

[12] Walid A. Hanafy, Qianlin Liang, Noman Bashir, Abel Souza, David Irwin, and Prashant Shenoy. Going green for less green: Optimizing the cost of reducing cloud carbon emissions. *ACM Digital Library*, 2024.

[13] Sofia Herelius. Green coding. can we make our carbon footprint smaller through coding? *BTH Blekinge Tekniska Högskola*, 2022.

[14] Merelo Guervos Juan Julián, Mora García Antonio Miguel, and García-Valdez Mario. Best practices for energy-thrifty evolutionary algorithms in the low-level language zig. *Universidad de Granada*, 2024.

[15] David Kacs, Joseph Lee, Justs Zarins, and Nick Brown. Pragma driven shared memory parallelism in zig by supporting openmp loop directives. *ArXiv*, 2024.

[16] Arya Kashyap. The adoption of green programming languages as a promising approach to improve computing's sustainability. *IJHSR International Journal of High School Research*, 2024.

[17] Taha Khudher. Green coding. energy consumption in different programming languages and algorithms. *DiVA Digitala Vetenskapliga Arkivet*, 2024.

[18] Dimitrios Koemtzopoulos, Georgia Zournatzidou, and Nikolaos Sariannidis. Can cryptocurrencies be green? the role of stablecoins toward a carbon footprint and sustainable ecosystem. *MDPI*, 2025.

[19] Patrick Kurp. Green computing. *Communications of the ACM*, 51(10):11–13, 2008.

[20] Christos P. Lamprakos, Lazaros Papadopoulos, Francky Catthoor, and Dimitrios Soudris. The impact of dynamic storage allocation on cpython execution time, memory footprint and energy consumption: An empirical study. *Research Gate*, 2022.

[21] S. J. Mahmudova. Green coding in programming and practices. *JET Journal of Engineering and Technology*, 2024.

[22] Niccolò Marini, Leonardo Pampaloni, Filippo Di Martino, Roberto Verdecchia, and Enrico Vicario. Green ai: Which programming language consumes the most? *arXiv*, 2024.

[23] Sergio Di Meglio and Luigi Libero Lucio Starace. Evaluating performance and resource consumption of rest frameworks and execution environments: Insights and guidelines for developers and companies. *IEEE Xplore*, 2024.

[24] Oyile Paul Oduor and Wabwoba Franklin. The evolution of green computing: Current practices and societal implications. *WJAETS World Journal of Advanced Engineering Technology and Sciences*, 2024.

[25] Kaya Oğuz. Achieving sustainable software systems by reducing bloat and by promoting green practices in software engineering education. *IEEE Xplore*, 2024.

[26] Pedro Pereira, Paulo Mendes, João Vitorino, Eva Maia, and Isabel Praça. Intelligent green efficiency for intrusion detection. *arXiv*, 2024.

[27] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages. how does energy, time, and memory relate? *ACM Digital Library*, 2017.

[28] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Elsevier, Science of Computer Programming*, 2021.

[29] Elisa Jiménez Riaza, Alberto Gordillo, Coral Calero, Mª Angeles Moraga, and Felix Garcia. Does the compiler version influence the energy consumption of programming languages? *SSRN Social Science Research Network*, 2024.

[30] Nicolas van Kempen, Hyuk-Je Kwon, Dung Tuan Nguyen, and Emery D. Berger. It's not easy being green: On the energy efficiency of programming languages. *ArXiv, Cornell University*, 2024.

[31] zihao jiao, Xiaoxin Xie, Mengyi Sha, and Wei Qi. Toward resilient green cloud computing: Joint operations of energy storage and spatial task allocation. *SSRN Social Science Research Network*, 2024.