Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Carbon-aware Software Services through Runtime Adaptation and Time Shifting

Ornela Danushi

ornela.danushi@phd.unipi.it

University of Pisa

Jacopo Soldani

University of Pisa

Stefano Forti

University of Pisa

Federico Giusti

University of Pisa

Antonio Brogi

University of Pisa

Research Article

Keywords:

Posted Date: November 18th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10383728/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Carbon-aware Software Services through Runtime Adaptation and Time Shifting

Ornela Danushi, Jacopo Soldani, Stefano Forti, Federico Giusti, Antonio Brogi

Department of Computer Science, University of Pisa, Largo B. Pontecorvo 3, Pisa, 56127, PI, Italy.

*Corresponding author(s). E-mail(s): ornela.danushi@phd.unipi.it;

Abstract

The increased environmental impact of the ICT sector calls for novel methodologies and tools to make software engineering environmentally sustainable. Software services should be designed to minimise their carbon footprint, without compromising on quality of service experienced by users. In particular, software services should inherently feature carbon-awareness by exploiting diverse operational configurations to reduce emissions without violating quality of service constraints. This work addresses this need via a dynamic adaptation of carbon-aware software services. In this article, we propose an optimisation scheme that dispatches the execution of incoming requests among distinct implementation of software services, minimising the overall carbon emissions through the combination of service adaptability and request time-shifting. To demonstrate the feasibility of our proposal, we provide an architectural solution and open-source prototype implementation, called Carbonshift. Controlled experiments, both on the lifelike validation of such scheme and on computationally intensive workloads, demonstrate that Carbonshift improves alternative solutions in reducing carbon emissions while keeping the required quality of service of the output results below a predefined threshold.

Keywords: Carbon Efficiency, Software Engineering, Sustainable Computing, Environmental Sustainability

1 Introduction

With software playing a key and increasingly active role, Information and Communication Technologies (ICT) are currently responsible for an estimated 2% of global carbon emissions (Belkhir and Elmeligi 2018; Malmodin et al. 2024; Mingay and Pamlin 2008). Thus, the interest in *sustainable software engineering* and, more specifically, *carbon-aware computing*, has grown significantly, focussing on reducing carbon emissions on each of the various stages of the software development life cycle (Danushi et al. 2025), i.e., from designing to decommissioning.

This entails considering the carbon emissions associated with software execution, for instance, by designing software services that dynamically adapt their behaviour to reduce runtime energy consumption and, consequently, mitigate environmental impact. According to the Software Carbon Intensity specification (Green Software Foundation 2024), the operational carbon emissions C (expressed in gCO_2-eq) of a running software service can be approximated as $C = E \times I$, where E denotes the energy consumed (in kWh) by the underlying infrastructure to operate the service, and I denotes the carbon intensity (in gCO_2-eq/kWh) of the energy mix powering such infrastructure. Notably, since carbon intensity fluctuates over time, executing the same software at different moments can possibly result in significantly varying carbon emissions, even if the actual software's energy consumption does not change.

Traditional energy-aware software design has operated under the concept that reducing energy consumption reduces carbon emission. Nevertheless, further reductions in carbon emissions can be achieved by considering how software (and its management) can adapt its behaviour (or its deployment) based on the carbon intensity of the energy grid (Aiello et al. 2024; Gsteiger et al. 2024). Consequently, recent research is focussing more on carbon-aware software adaptation, wherein systems dynamically realign their execution profiles to coincide with periods of low carbon intensity of the energy supply (Danushi et al. 2025; Angiolini et al. 2026). Said otherwise, the goal should be consuming less energy during high-emission periods, running most high-quality computation during low-emission periods.

Building upon this idea, we aim at enabling software developers to implement multi-versions of software services, either from scratch or by systematic refactoring, so as to inherently support carbon-aware execution. Our methodology incorporates a principle from approximate computing (Mittal 2016), wherein execution time reductions-and consequently energy saving-are achieved by tolerating approximate outputs with higher error rates. Aligning with our previous work (Angiolini et al. 2026), we formalise the design of software services into multiple implemented versions, called *flavours*. While each flavour maintains functional equivalence, they exhibit distinct execution times and Quality of Service (QoS) trade-offs, typically quantified via output error rates. For instance, a machine learning task is executed by a service that is implemented in three different flavours (viz., algorithms): *high* (averaging a 20-second execution time with 0% error), *medium* (averaging 10 seconds with 3% error), and *low* (averaging 1 second with 6% error).

Rather than choosing only one flavour *a priori*, our proposal enables developers to implement these software versions as decoupled execution *strategies*, leveraging the *Strategy* design pattern as exemplified by Angiolini et al. (2026). This enables

the service context to dynamically arbitrate between strategies at runtime, adapting to real-time fluctuating carbon intensity, so as to minimise carbon emissions while constraining the average error within a predefined target threshold. Nevertheless, a limitation of this reactive selection approach is its inability to account for task deferrability. By incorporating a temporal shifting mechanism for deferrable tasks to periods of lower carbon intensity, further reduces emissions while potentially improving overall QoS, i.e., serve deferrable tasks with a more energy-intensive service flavour when cleaner energy is available.

To address this gap, this work also builds on the *Strategy* design pattern by introducing a novel optimisation scheme to enable carbon-aware services selection with time-shifting strategies. Such selection evaluates the feasibility of shifting incoming service requests to periods characterised by lower carbon intensity while simultaneously maintaining an average output quality. By accounting for user-specified deadlines on service requests, informed time-shifting decisions are thus made when needed/possible. Formalised as an Integer Linear Programming problem, our optimisation scheme determines the best trade-off among carbon-emissions and overall service error. Specifically, the optimisation scheme determines *when* and with *which strategy* to serve incoming requests. The objective function minimises total carbon emissions while ensuring that deadline constraints are respected for each request and that the aggregate service error remains bounded below a user-specified error threshold, i.e., the user's preference regarding the average quality of requests permitted over a maximum time window.

To showcase the feasibility of our proposal, we introduce an open-source *prototype* tool, called Carbonshift, and an architectural solution, which enable running our optimisation scheme in practice. Carbonshift was also used to run *controlled experiments* and distinct *case studies* aimed at assessing the effectiveness and robustness of our proposal under real executions of computationally intensive workloads. Namely, we simulated a realistic scenario with a multi-flavoured service receiving varying amounts of requests, which arrive and get served over time, with fluctuating values of carbon intensity. Our experiments compare our approach with other possible policies for choosing when to serve incoming requests and with which strategy. The results of the experiments show that, among the policies that kept the overall service error within a set error threshold, ours was the one also resulting in the lowest overall carbon emissions. We aim at showing that, among the policies that keep the overall service error within a set error threshold, ours achieves the most significant reduction in overall carbon emissions.

The rest of this article is organised as follows. Section 2 provides an example motivating the need for our proposal. Section 3 introduces the proposed optimisation scheme, which is then implemented by Carbonshift (Section 4.1) and designed as an architectural solution (Section 4.2). Section 5 presents our controlled experiments and evaluations on distinct case studies. Finally, Sections 6 and 7 discuss related work and draw some concluding remarks, respectively.

This article extends our previous work (Danushi et al. 2026) by introducing an architectural solution that enables realising carbon-aware software services in practice

(Section 4.2) and by assessing the effectiveness of our overall proposal by means of controlled experiments and case studies (Section 5). The architectural solution, controlled experiments, and case studies are brand new and first presented in this article.

2 Motivating Example

Let a service enacting a classification task via machine learning models be available with three alternative model-based execution strategies. The strategy *high* achieves a 0% error in the classification outputs (i.e., 100% accuracy) while requiring an average execution time of 20 s on the available runtime, on average. Conversely, the strategy *low* produces results with an average error of 6% while reducing the average execution time to 1 s. Finally, the strategy *medium* represents a trade-off between the former two, resulting in a 3% error and 10 s execution time, on average. Incoming requests toward such service arrive over contiguous time slots, and we must guarantee that each request is served within its designated deadline.

We want to minimise carbon emissions, while ensuring that the cumulative average error remains below the 5% threshold deemed acceptable for this scenario's business needs. To this end, we aim at determining *when* and via *which strategy* to serve each incoming request, by evaluating both the current and forecast carbon intensity of the energy powering the server where our service runs during each time slot. The operational carbon emissions for serving a request are estimated as the product of the slot's carbon intensity (in gCO_2-eq/kWh) and the energy consumed by the server to respond to the request (in kWh). The latter is computed as the multiplication of the server's power consumption (e.g., $50W$) and the execution time of the selected strategy (Green Software Foundation 2024).

Figure 1 illustrates two different scenarios. Particularly, incoming requests are represented as boxes annotated with their deadlines and they are placed in the table at the intersection between the selected strategy and timeslot for their execution. In both scenarios, we assume that 10 requests arrived with the following deadlines: three must be served immediately (i.e., within the current slot), one that can be postponed to the next slot, and six can be delayed by up to two time slots.

An example of a greedy strategy-selection policy is the *cleanest-slot* policy, which serves each request using the strategy *high* in the greenest available time slot within the request's deadline. In the considered scenario (Figure 1a), this request allocation yields an overall 0% error and total carbon emissions of 0.2111 gCO_2-eq. This total is obtained by summing the emissions from Slot 0 (0.1111 gCO_2-eq) and Slot 2 (0.1000 gCO_2-eq). Specifically, the three requests that require immediate execution are handled in Slot 0, while all deferrable requests are postponed to Slot 2, with all requests being served via the strategy *high*.

Alternatively, another greedy strategy selection policy is the *error-driven* policy, which selects to serve a request with the strategy based on the cumulative average error accumulated up to the current slot. Specifically, if the average error exceeds the designated threshold, the current request is served using the strategy *high*; otherwise, a strategy is selected at random. In the considered scenario (Figure 1b), this request allocation yields an overall error below 5% and total carbon emissions of 0.1555

Strategy	Error	Duration	Slot 0	Slot 1	Slot 2
	%	s	100 gCO_2-eq/kWh	119 gCO_2-eq/kWh	60 gCO_2-eq/kWh
High	0	20	0, 0, 0, 1		2, 2, 2, 2, 2, 2
Medium	3	10			
Low	6	1			

(a)

Strategy	Error	Duration	Slot 0	Slot 1	Slot 2
	%	s	100 gCO_2-eq/kWh	119 gCO_2-eq/kWh	60 gCO_2-eq/kWh
High	0	20	0, 2, 2		
Medium	3	10	0, 1, 2, 2, 2		
Low	6	1	0, 2		

(b)

Fig. 1: Requests allocation across time slots with known carbon intensities by selecting strategies characterised by specific errors and durations. The *Cleanest-slot* policy (a) shifts requests to the cleanest slot, while the *Error-driven* policy (b) combines strategies to meet the cumulative error threshold of 5% within the first available slot.

gCO_2-eq. This total is obtained by summing the Slot 0 emissions from the strategy *high* (0.0833 gCO_2-eq), *medium* (0.0694 gCO_2-eq), and *low* (0.0028 gCO_2-eq).

The *cleanest-slot* and *error-driven* greedy policies are designed by exploiting different dimensions of the search space. Nevertheless, such policies are inherently suboptimal, as they focus on only one optimisation dimension at a time, i.e., either altering the serving strategy or shifting its processing in time. To address this limitation, in the next section we propose an optimisation scheme that jointly considers both dimensions to identify the most convenient trade-off between carbon emissions and overall service error.

3 Carbon-Aware Request Scheduling and Adaptation

In this section, we first illustrate our optimisation scheme to perform both carbon-aware strategy selection and time shifting (Section 3.1). We then retake our motivating example to illustrate how our proposal works (Section 3.2).

3.1 Problem Formulation

To dynamically map incoming service requests to specific strategies and time slots, while leveraging forecast carbon intensity and desired error thresholds, we propose a mathematical formulation. The objective of this formulation is to minimise carbon emissions while ensuring that the average error of the output results is upper-bounded by a tolerated maximum error threshold.

Problem Variables Consider parameters and decision variables summarised in Table 1. Let R denote the set of requests arriving at time slot 0, each characterised by a specific deadline representing the maximum number of time slots within which it must be served. Since R may scale to hundreds of thousands requests, allocating each one individually would introduce an excessive number of decision variables, making

Table 1: Problem parameters.

Symbol	Meaning
R	Set of requests
β	Number of blocks of requests
B	Set of all blocks
b	Block b of requests
l_b	Deadline of block b
Δ	Number of time slots
c_t	Estimated carbon emissions in time slot t
S	Set of available strategies
s	Strategy s
d_s	Average duration for strategy s
e_s	Average error for strategy s
ε	Tolerated maximum error

the problem computationally intractable. Thus, the $|R|$ requests are partitioned into β distinct request blocks, each containing the same amount of requests. A block of requests is then assigned to a time slot and strategy, which will used for processing all the requests it contains.[1]

To approximate the original deadline distribution of set R, requests are first sorted by deadline and then clustered into β blocks of requests. If a block contains requests with heterogeneous deadlines, it is assigned the minimum deadline l_b of its requests. Given the resulting set of blocks B, each block $b \in B$ is scheduled to be served within its deadline l_b, which indicates the maximum number of time slots allowed to serve all requests within b.

Our optimisation scheme determines *when* the requests within each block $b \in B$ should be served. This scheduling occurs within a time window that starts at the arrival time slot and spans Δ discrete time slots, where Δ can be configured to customise the request allocation problem. The forecast carbon intensity for each time slot $t \in \{0, 1, \ldots, \Delta - 1\}$ is denoted by c_t, and can be sourced from external services such as the daily forecasts provided by Carbon Intensity UK.[2]

Additionally, given the set of available execution strategies S, the optimisation scheme selects *which strategy* $s \in S$ to serve the requests within a block $b \in B$. This selection accounts for the average error e_s and execution time d_s associated with each strategy $s \in S$. Finally, since the objective is to assign a strategy and a time slot to each request block to minimise overall carbon emissions, we denote the assignment of block b to strategy s at time slot t as x_{tsb}.

Constraints The decision variable x_{tsb} is set to 1 if request block b is served using strategy s in time slot t, and 0 otherwise (Eq. 1).

$$\forall b \in B. \quad \forall s \in S. \quad \forall t \in [0, \Delta) : x_{tsb} \in \{0, 1\} \tag{1}$$

[1] This does not cause any loss of generality, as setting $\beta = |R|$ results in solving the most general problem of allocating each request separately.

[2] https://carbonintensity.org.uk

Each block $b \in B$ must be assigned to exactly one strategy and exactly one time slot (Eq. 2). Additionally, the selected time slot must fall within both the block's deadline and the defined time window (Eq. 3).

$$\forall b \in B : \sum_{s \in S, t \in [0, \Delta - 1]} x_{tsb} = 1 \tag{2}$$

$$\forall b \in B : \sum_{s \in S, t \in [0, \min\{l_b, \Delta - 1\}]} x_{tsb} = 1 \tag{3}$$

Note that when $l_b \geq \Delta - 1$, both Equation (2) and Equation (3) become equivalent. Conversely, when instead $l_b < \Delta - 1$, they jointly enforce the following constraint:

$$\sum_{s \in S, t \in [0, l_b]} x_{tsb} = 1 \qquad \wedge \sum_{s \in S, t \in [l_b + 1, \Delta - 1]} x_{tsb} = 0.$$

Additionally, the maximum tolerated error ε must be respected (Eq. 4).

$$\sum_{b \in B,\ s \in S,\ t \in [0, \Delta - 1]} x_{tsb} \cdot e_s \leq \varepsilon \cdot \beta \tag{4}$$

Objective Function The objective is to assign request blocks to time slots and strategies that minimises total carbon emissions, as the constraints ensure the average error respects the global tolerance threshold ε. Namely, our objective function is:[3]

$$\min \sum_{b \in B,\ s \in S,\ t \in [0, \Delta - 1]} x_{tsb} \cdot c_t \cdot d_s$$

3.2 Motivating Example Retaken

We demonstrate how the proposed optimisation scheme minimises carbon emissions while satisfying the tolerated error threshold. As in Section 2, we consider 10 incoming requests with deadlines $[0, 0, 0, 1, 2, 2, 2, 2, 2, 2]$. These requests are clustered into $\beta = 5$ blocks, i.e., $\boxed{0}$, $\boxed{0}$, $\boxed{2}$, $\boxed{2}$, $\boxed{2}$, each containing 2 requests per block[4] (e.g., the first box $\boxed{0}$ contains the first two requests, i.e., $[0, 0]$).

Figure 2 shows the allocation resulting from applying our optimisation scheme. A total carbon emission of 0.0405 gCO_2-eq is obtained by summing the contributions from Slot 0 (0.0055 gCO_2-eq) and Slot 2 (0.0349 gCO_2-eq), while maintaining an average error of 4.8%. Compared to the *cleanest-slot* (0.2111 gCO_2-eq) and *error-driven* (0.1555 gCO_2-eq) naïve policies (Figure 1), our proposal reduces total emissions by approximately 80.81% and 73.95%, respectively. These results intuitively show how our proposed scheme not only minimises carbon emissions but also controls the error

[3] For simplicity, we abstract from execution details (e.g., slot capacities), assuming that each assigned request block can be fully processed within its designated time slot and strategy.

[4] When requests within a block exhibit heterogeneous deadlines, the block is assigned the minimal deadline among them, which satisfies all requests.

Strategy	Error %	Duration s	Slot 0 100 gCO_2-eq/kWh	Slot 1 119 gCO_2-eq/kWh	Slot 2 60 gCO_2-eq/kWh
High	0	20			
Medium	3	10			2, 2
Low	6	1	0, 0		2

Fig. 2: Request blocks, denoted as boxes, are allocated across time slots with known carbon intensities by selecting strategies characterised by specific errors and durations, thereby ensuring that the cumulative error remains below a 5% threshold.

parameter, thereby achieving a more balanced trade-off between environmental impact and service QoS.

4 Enabling Carbon-Awareness, in Practice

We show our open-source command-line prototype implementation (Section 4.1), built upon the previously presented optimisation scheme formalisation (Section 3). Then, we design an architectural overview enabling such prototype implementation to serve asynchronous software service requests autonomously (Section 4.2).

4.1 Prototype Implementation

Our prototype implementation, called Carbonshift,[5] is developed in Python and runs by issuing the command in Figure 3a. Carbonshift takes three input files: `input_requests`, i.e., integer values defining the deadlines of the input requests, in a single line; `input_strategies`, i.e., rows of integer values defining the name, error, and duration of each implemented strategy; and `input_co2`, i.e., Δ rows of integer values indicating carbon emission intensities of each time slot. Furthermore, also configurable parameters are taken as input: `DELTA` limits the window size Δ; `BETA` approximates requests by optionally defining the number of request blocks β (by default, $\beta = 1000$); `ERROR` defines the maximum allowable average error, denoted by ε. Finally, the `output_assignment` file contains the output of the service allocation results produced for the given input scenario.

Figure 3b illustrates the output assignment for our motivating example (Section 2), as computed by Carbonshift. Each row contains the per-request assignment, i.e., the request identifier, the name of the strategy selected to serve such request, the chosen execution time slot, the estimated carbon emissions, and the error associated with the selected strategy. Carbonshift not only shows the per-request assignment, it also provides information on the overall carbon emissions of the assignment itself (Figure 3c). Specifically, it details both *all_emissions* as the total carbon footprint generated by serving all incoming requests, and *slot_emissions* as carbon emissions per individual time slots (i.e., the entry at the i-th position represents the total emissions due to serving all requests specifically to slot i).

[5] Carbonshift is publicly available at: https://github.com/di-unipi-socc/carbonshift.

```
python3.8 carbonshift.py
    input_requests
    input_strategies
    input_co2
    DELTA
    BETA
    ERROR
    output_assignment
```

(a)

```
0,Medium,2,0.008333,3
1,Medium,2,0.008333,3
2,Medium,2,0.008333,3
3,Low,0,0.001389,6
4,Medium,2,0.008333,3
5,Low,0,0.001389,6
6,Low,2,0.000833,6
7,Low,0,0.001389,6
8,Low,0,0.001389,6
9,Low,2,0.000833,6
```

(b)

```
all_emissions:
    0.04055400000000001
slot_emissions:
    [0.005556, 0, 0.034998]
```

(c)

Fig. 3: Command-line invocation (a), and output of request assignment (b) and aggregated statistics (c) returned by Carbonshift on our motivating example.

Carbonshift computes the request allocation by using a CP-SAT solver, built upon Google's OR-Tools optimisation suite.[6] Alternative implementations of the solver and optimisation approaches (e.g., dynamic programming), from a preliminary scalability analysis, resulted more inefficient. We plan to run a more comprehensive evaluation of the potential implementations for future work.

4.2 Architecting Carbon-Aware Software Services

We design an architectural solution to achieve carbon-emission reductions by serving requests of alternative software service implementations. Such solution is scalable, as it can process variable volumes of requests; customisable, by allowing the specification of different assignment policies; and modular, thereby facilitating seamless integration among components. Furthermore, temporal signals automatically trigger events via a publish-subscribe pattern, thereby guaranteeing decoupling among components.

The architectural components building our proposal are organised in a cyclic way (Figure 4). The Client sends requests to the Frontend API component of our service, which forwards them to the Ingress Queue. The latter is an asynchronous message queue implemented via RabbitMQ (Broadcom 2026). At each time slot, the Scheduler dequeues requests from the Ingress Queue, applies the configured assignment policy, and forwards them – along with metadata regarding the current execution time slot and the selected serving strategy – to another message queue, the Slot Queue. The duration of each time slot is regulated by an external Global Clock, which also triggers the Service to dequeue requests from the Slot Queue. More precisely, during slot i, the Service dequeues requests corresponding to the previous slot *i-1*, serves them according to the selected strategy, and sends a callback notification back to the Client.

The final metadata of the served requests includes the *input task*, the *exit task characteristics*, the selected *strategy*, the executing *time slot*, and the resulting *error* produced by the assignment policy. While the first two fields are used to mark the client-request as completed, the remaining fields are leveraged for statistical evaluations of the assignment policy. For instance, the overall carbon emissions generated by

[6] https://developers.google.com/optimization

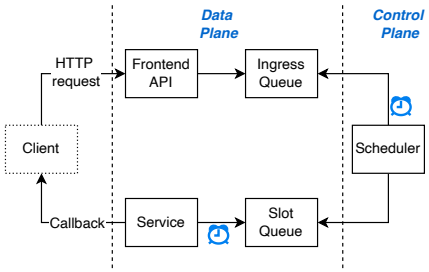


Fig. 4: Our proposed service architectural solution. Arrows model the flow for serving an incoming request.

the assignment policy itself are recorded and the emissions due to serving the requests assigned to each slot (i.e., *all_emissions* and *slot_emissions* field in Section 4.1).

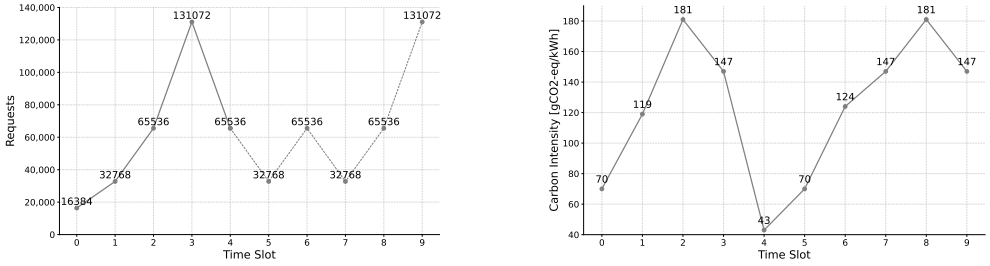
5 Experimental Assessment

Given the lack of solutions tackling the same problem as Carbonshift (see Section 6), and to avoid an unfair comparison by adapting state-of-the-art solutions to work for different objectives, we compare Carbonshift against several strategy selection policies, which represent possible heuristics that embody different trade-offs for balancing the objectives of either reducing carbon emissions or respecting error thresholds within deadline constraints. The considered policies are:

- *Baseline* (with *high* strategy, no time shift): Each request is served in the slot it arrives with the strategy with the lowest error, i.e., *high*,
- *Random* strategy (with no time shift): Each request is served in the slot it arrives with a randomly selected strategy,
- *Carbon-driven* strategy (with no time shift): It serves all the current slot's requests with a strategy chosen based on the slot's carbon intensity, i.e., *high* for green slots, *medium* for yellow slots, and *low* for brown slots,[7]
- *Error-driven* strategy (with no time shift): It chooses the strategy for serving a request based on the average error accumulated in the current slot, i.e., if the average error is above the given threshold, it serves the current request with strategy *high*, otherwise it randomly selects one of the possible strategies, and
- *Cleanest-slot* (with *high* strategy): Each request is served with the strategy *high* in the slot with the lowest carbon intensity among those from the arrival time slot and the last possible slot according to the request's deadline.

Our evaluation is driven by two primary objectives. First, to assess the effectiveness of the proposed optimisation scheme in reducing carbon emissions while meeting error thresholds and deadline constraints, we present a comparative analysis within a simulated controlled experiment (Section 5.1). Second, to evaluate the robustness of the

[7]Based on data available on 27th May 2025, a slot is classified as green if its carbon intensity is below 119 gCO_2-eq/kWh, yellow if it is between 119 and 180 gCO_2-eq/kWh, and brown if it is higher than 180 gCO_2-eq/kWh.



(a) (b)

Fig. 5: Simulation setup: incoming requests (a) and carbon intensity forecasts (b) per time slot.

scheme under real executions of computationally intensive workloads (e.g., machine learning tasks), we examine its performance through distinct case studies (Section 5.2). Finally, we discuss the potential threats that may apply to the validity of our study (Section 5.3).

5.1 Controlled Experiment

Setup. We simulated a realistic scenario in which the application in our motivating example (Section 2) receives a varying amount of incoming requests, which arrive and get served over time, with fluctuating values of carbon intensity. The experiment consists of exploiting our proposal and alternative policies to select the time slots and strategies used to serve the incoming requests and in computing the carbon emissions that would have resulted from their actual execution.

The simulation[8] spans five contiguous time slots, namely the time slots 0–4 in Figure 5, for each of which we plan the allocation of the incoming requests with $\Delta = 5$ and at varying error thresholds $\varepsilon \in \{2\%, 4\%, 5\%\}$. Figure 5 shows (a) the considered incoming request volumes, which span between 2^{14} and 2^{17} requests to have a large-scale benchmarking and (b) fluctuating values of carbon intensity.[9]

Results. The results of our experiments[10] are summarised in Figure 6, which shows the average error and reduction of carbon emissions of Carbonshift (with $\beta = 1000$) against those of the alternative strategy selection policies. The percentages shown in the figure are computed against the *Baseline*, i.e., serving all requests in the arriving time slot with the *high* strategy.

By looking at Figure 6, we can observe that Carbonshift steadily provides the better trade-off among reduction of carbon emissions and average error, with the latter always being aligned with the tolerated error threshold. The same does not hold for the other policies, which result in less reduction of carbon emissions, sometimes even providing results whose average error is above the tolerated error threshold (i.e., *Random* and

[8]The simulation was run on a machine equipped with an AMD Ryzen 7 5800H CPU @ 3.2 GHz (8 cores / 16 threads) and 32 GB of RAM.
[9]Based on data available at: https://carbonintensity.org.uk, we consider carbon emissions following a double inverted U-shaped trend.
[10]All experimental results are available at: https://short.do/wbZM_Y.

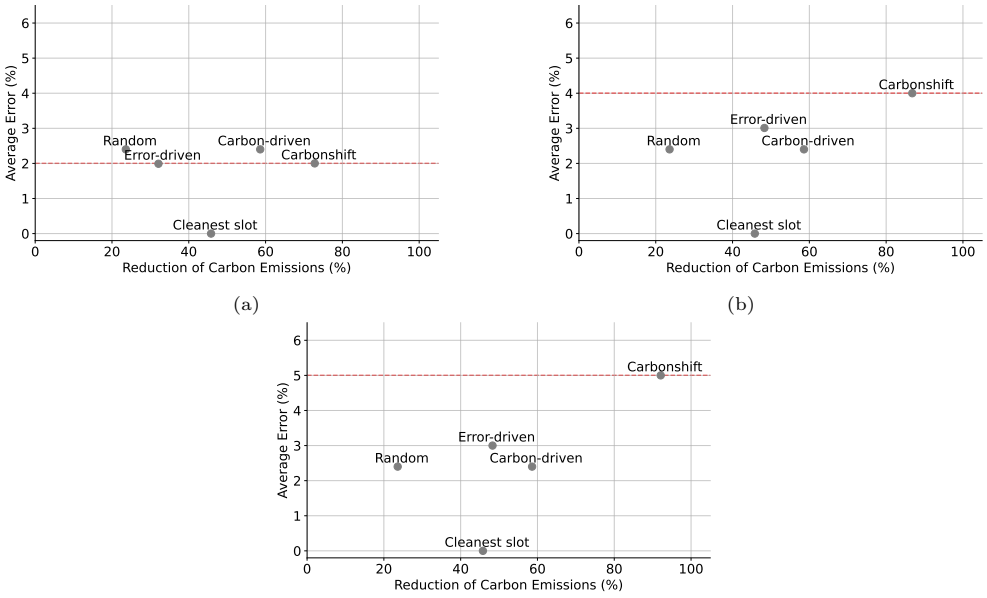


Fig. 6: Carbon-emissions reductions with respect to the *Baseline* policy and average error, with threshold ε set to (a) 2%, (b) 4%, and (c) 5%. The rationale for the chosen values of ε is to analyse how different selection policies find different compromises between the available strategies, i.e., *high* (error 0%), *medium* (error 3%), and *low* (error 6%).

Carbon-driven in Figure 6a). This happens mainly because the considered alternative strategy selection policies focus on only one of the two dimensions that are instead jointly optimised by Carbonshift, i.e., average error and reduction of carbon emissions.

From Figure 6, we can also observe that, the higher the tolerated error threshold, the more effective is Carbonshift in reducing carbon emissions. Particularly, from a reduction of 72.8% by setting the error threshold $\varepsilon = 2$ (Figure 6a), Carbonshift can achieve a reduction of 92.1% with $\varepsilon = 5$ (Figure 6c).

5.2 Case studies

Setup. We run three case studies based on three different machine learning Natural Language Processing (NLP) tasks as in Angiolini et al. (2026). Specifically, we opted for text generation, named entity recognition, and question answering, due to their wide adoption and differentiation in terms of computational complexity. Each service was evaluated using three different model implementations from Hugging Face (2026), characterised by varying execution times and output error levels (Figure 7).

To estimate the execution times, ten benchmark runs were performed with predefined input parameters. Meanwhile, the average error thresholds (ε in Figure 8d) were set to increasing fixed integer values close to those of the models (*error* in Figure 7).

strategy	*time*	*error*	*model*
High	27.0969 s	13.10 %	EleutherAIgpt-neo-1.3B
Medium	19.4288 s	17.50 %	gpt2-xl
Low	11.7405 s	22.10 %	gpt2-large

(a) Text Generation

strategy	*time*	*error*	*model*
High	4.6624 s	2.50 %	Jean-Baptiste/roberta-large-ner-english
Medium	4.5252 s	8.30 %	tner/roberta-large-conll2003
Low	4.3850 s	8.70 %	dslim/bert-base-NER

(b) Named Entity Recognition

strategy	*time*	*error*	*model*
High	12.6052 s	6.85 %	bert-large-uncased-whole-word-masking-finetuned-squad
Medium	6.2079 s	8.20 %	deepsetroberta-base-squad2
Low	2.6776 s	13.00 %	distilbert-base-cased-distilled-squad

(c) Question Answering

Fig. 7: Natural-language-processing task specifications on service evaluations and model implementations.

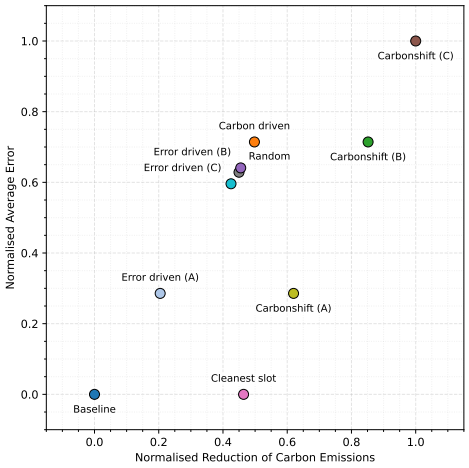
Intuitively, configuration A considers the tightest ε values (e.g., 15% for text generation, 3% for named entity recognition, 7% for question answering, given the average *error* value of 13.10%, 2.50%, 6.85%. Conversely, configuration C considers the broadest ε values as error threshold, and configuration B represents a trade-off between the former two.

Specifically, the text-generation task aims at generating text from a given prompt (e.g., input: "Climate change affects", output: "Climate change affects ecosystems, weather patterns, and human activities."). The metric used for evaluating the performance of the text-generation models is the Perplexity on WikiText-103 benchmark (Merity et al. 2016). The named-entity-recognition task aims at the identification and classification of named entities within a text (e.g., input: "Apple will release a new iPhone in San Francisco", output: "Apple [ORGANIZATION], iPhone [PRODUCT], San Francisco [LOCATION]"). The metric used for evaluating the performance of the named-entity-recognition model is the F1 Score. Finally, the question-answering task aims at extracting answers to questions based on a textual context (e.g., context input: "Albert Einstein was a theoretical physicist who developed the theory of relativity.", question input: "Who developed the theory of relativity?", output: "Albert Einstein"). The metric used for evaluating the performance of the question-answering model is the F1 Score.

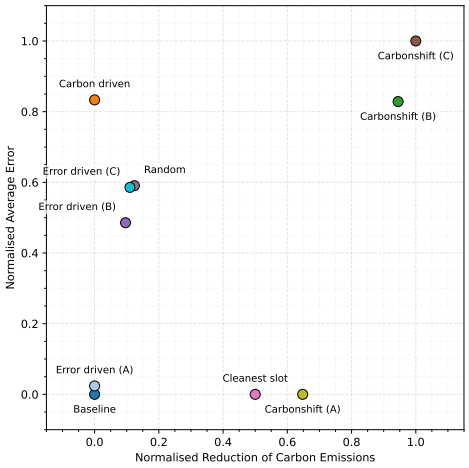
For each task, we define the same pre-defined load of 35 requests, with $\beta = 35$ to allocate each request individually. The simulation[11] spans five contiguous time slots, each associated with a carbon intensity value (e.g., 100, 90, 120, 80, 105 of gCO_2-eq/kWh). The experiment consists of exploiting our architectural proposal to test Carbonshift and the alternative policies to select the time slots and strategies used to serve the incoming machine learning task requests.

Results. The results of our experiments are summarised in Figure 8, which shows the min-max normalised average error and carbon emissions of Carbonshift against those of the alternative strategy selection policies. Policies closer to the top left of the

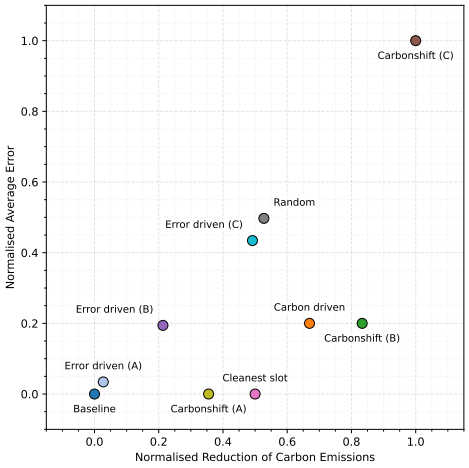
[11]The simulation is run on a MacBook Pro 14" (2021) host machine, equipped with an Apple M1 Pro (8-core CPU, 14-core GPU) and 16 GB of unified memory.



(a) Text Generation



(b) Named Entity Recognition



(c) Question Answering

task	$\varepsilon(A)$	$\varepsilon(B)$	$\varepsilon(C)$
Text Generation	15%	18%	20%
Named Entity Recognition	3%	8%	9%
Question Answering	7%	8%	12%

(d) Threshold Configurations

Fig. 8: Min-max normalised values of policy carbon emissions and average error. When the average error threshold is binding, are selected *base*-clustering configurations of ε values (e.g., A the tightest, C the broadest) with the aim of analysing how different selection policies compromise between the available strategies and increasing progressive average error thresholds.

Cartesian plane represent the worst trade-off between reductions of carbon emissions and average error thresholds.

The *Baseline* and *Random* policies allocate requests without considering the carbon intensity of the available slots. While the *Random* policy exhibits unpredictable behaviour, the *Baseline* yields the highest carbon emissions alongside the lowest average error. This outcome is expected due to its systematic selection of the most accurate

and computationally demanding strategy. The *Cleanest-slot* policy results in fewer carbon emissions.

The *Carbon-driven* allocation of requests with a strategy selection depends on the carbon intensity of the currently executing slot, e.g., lower reductions in question-answering task (Figure 8c) and higher reductions in named-entity-recognition task (Figure 8b). The *Error-driven* policy exhibits similar behaviour across progressive *base*-clustering configurations for named-entity-recognition task (Figure 8b) and question-answering task (Figure 8c); however, for text-generation task (Figure 8a), a switch occurs between *configuration B* and *configuration C* which may be attributed to the randomness within its evaluations.

In all settings, Carbonshift satisfies all user constraints and significantly reduces carbon emissions in a controlled manner, i.e., avoiding excessive QoS degradation. Naturally, the greater the ε values for the average error thresholds, the better the optimal trade-off that can be achieved.

5.3 Threats to Validity

Based on the taxonomy developed by Wohlin et al. (2000), four potential threats may apply to the validity of our study.

Construct validity. We quantify environmental impact as estimated operational carbon emissions and QoS as average output error. Emissions are computed from the carbon intensity of each time slot and the execution time of the selected strategy under the duration-based energy model introduced in Section 2; hence, power fluctuations, carbon-intensity forecast errors, orchestration overheads, and embodied emissions are not captured. Likewise, a fixed strategy-level error abstracts from input-specific and user-perceived output quality. We limit these threats by following the SCI relation, adopting task-specific Perplexity and F1 metrics, and estimating model execution times over ten benchmark runs.

Internal validity. Given the lack of solutions addressing the same objective, our comparison relies on five purpose-built policies, whose implementation – particularly the random choices of *Random* and *Error-driven* – may affect the observed differences. To limit this threat, all policies are run within the same controlled scenarios, using common workloads, deadlines, carbon-intensity values, and strategy profiles on the same host. Request clustering is another source of approximation. However, sorting requests by deadline and assigning each block its minimum deadline conservatively preserves deadline constraints. Nevertheless, the selected β and Δ values and the assumption of exact forecasts may influence the resulting assignments.

External validity. Our assessment considers a short scheduling horizon, a small number of fixed request and carbon-intensity profiles, and three NLP tasks, each evaluated with 35 requests and three models on a single case-study platform. Results may therefore differ for longer horizons, other regions and hardware platforms, non-NLP services, or interacting services. We mitigate this threat by complementing a large-scale controlled experiment, spanning 2^{14}–2^{17} requests and three error thresholds, with real executions of three widely adopted NLP tasks having different computational complexities.

Conclusion validity. Our evidence is descriptive, then the observed reductions should not be interpreted as evidence of general statistical significance. We limit this threat by comparing against five policies across multiple error thresholds and tasks, reporting reductions against a common baseline in the controlled experiment, and making both the prototype and the controlled-experiment results available. Our conclusions are therefore restricted to the assessed lifelike settings: among the evaluated policies satisfying the modelled error and deadline constraints, Carbonshift yields the lowest estimated operational emissions.

6 Related Work

Previously, we investigated existing solutions of carbon-aware software design and development (Danushi et al. 2025). Typically, such solutions rely on *external, ad-hoc implementations* which provide carbon-efficient metrics, e.g., purposefully built APIs engineered by the Green Software Foundation (2025). Consequently, adaptive and flexible workload automatic management solutions have been used, also from an architectural perspective, such as the one of Google data centers (Radovanovic 2020). To advance with this view of being adaptable, Giussani (2026) aligns with our conceptualisation of "time shifting" by postponing compute tasks to coincide with periods of low grid carbon intensity. They define carbon-footprint metrics when deploying Software-as-a-Service applications, while we focus toward application-level self-adaptation. Specifically, we aim at developing software services that can adaptively change their flavour selection while serving incoming requests, based on the fluctuating carbon intensity of the targeted execution windows.

Alternative solutions to carbon-awareness focus on a control-loop orchestration trading-off between clean energy availability and QoS metrics. Specifically, Souza et al. (2023) propose a multi-objective optimisation scheme designed for the provisioning and carbon-aware scheduling of geo-distributed web services. Similarly, Hasan et al. (2016, 2017) use a Monitor Analyse Plan Execute Knowledge (MAPE-K) framework combined with Condition Event Action (CEA) loop. De Oliveira and Ledoux (2010) also apply the MAPE and CEA loop, specifically for Service Oriented Architectures. Extending this approach, Bhatt et al. (2025) integrate the MAPE-K loop with Decision Maps to monitor multifaceted runtime events-ranging from internal states and environmental shifts to energy consumption and data distribution-thereby triggering contextual adaptability. Whereas these solutions depend on externally provided services, our approach intrinsically enables the design and development of carbon-aware services to dynamically adapt themselves to different carbon intensities.

Beyond software-level control, alternative adaptive solutions focus on infrastructure-level resource substitution to reduce energy consumptions. Resource-aware strategies have been proposed to dynamically scale the execution environment of software services, either by balancing CPU and memory utilisation (Bunse et al. 2009; Bunse and Höpfner 2008) or by reconfiguring cloud resources for interactive and batch workloads (Xu et al. 2021). Similarly, Kavanagh and Djemame (2018) decompose applications into independent tasks managed across heterogeneous resources to reduce energy consumption. Despite targeting similar efficiency goals, their approaches

focus on adapting the runtime infrastructure, while our approach empowers software services to natively and dynamically self-adapt to reduce carbon emissions.

More similar to us, Gazza et al. (2026) rely on delivering services in multiple flavours through a constraint optimisation approach to deploy multi-flavoured services across cloud-edge infrastructures aiming at maximising both failure resilience and carbon efficiency. Our approach can be seen as complementary to Gazza et al. (2026). Indeed, whereas their focus is on *where* to orchestrate flavoured services to reduce carbon emissions, we determine *when* to process incoming requests. By managing execution dynamically, we enable services to process requests in the cleanest time slots with the flavour selection bounding the overall error within a given threshold.

We extend our previous approach (Angiolini et al. 2026), for interactive software services that require immediate execution. The previous framework relies on carbon intensity forecasts to assign a single, static flavour to each daily time slot, which is then applied uniformly to all incoming requests in that slot. We generalise this idea by targeting services whose incoming requests can be deferred within a given deadline. By relaxing the rigid assumption of a single flavour per slot, we introduce time-shifting capabilities and different flavour selections to be used to serve different requests.

Summing up, our approach builds upon and extends several key ideas from the existing literature. While the majority of prior efforts focus on energy consumption with only a few explicitly accounting for carbon intensity, our solution fully embraces carbon-awareness by operationalising the widely recognised principle of time shifting (e.g., through deferred request execution). Specifically, we trade-off performance aspects with carbon efficiency by leveraging multi-flavour service variants. Ultimately, by integrating these mechanisms with autonomous adaptation strategies, we enable a new class of purposely built services capable of dynamically adjusting their own behaviour to fluctuations of the carbon intensity of the power supply.

7 Conclusions

In this article, we presented an optimisation scheme that minimises total carbon emissions by both adapting service behaviour and performing service requests' time shifting based on the current workload and forecast carbon intensity of the energy grid. We also designed an architectural solution and introduced Carbonshift, an open-source prototype implementation of such scheme.

Preliminary experiments against alternative request allocation policies confirm that Carbonshift successfully computes the request allocation that results in the lowest overall carbon emissions, while strictly bounding the cumulative service error within user-defined thresholds. In addition, we evaluated Carbonshift across diverse case studies involving computationally intensive machine learning tasks. The results indicate that our prototype consistently achieves the most convenient trade-off in terms of carbon emission reductions and overall average error, i.e., greater average error the better in terms of reduced emissions.

For future work, we plan to devise a full-fledged support to design and developed multi-flavoured services, such as those targeted by our proposal. For example, we plan to support semi-automatically refactoring existing services towards multi-flavoured

services by automatically generating service variants, through e.g., the compiler-driven technique by Couroussé et al. (2016) or the LLM-based approach by Xia et al. (2025).

Another interesting direction is extending our approach to support multi-service applications, thus dealing with communication overheads, complex-service dependencies, and data exchange frequencies, which could significantly alter the overall application's carbon footprint. This would also enable experimenting our proposal in multi-service, industry-scale applications, which is also in the scope of our planned future work. Particularly, we plan to investigate how to fine-tune the extended problem parameters (e.g., the number of blocks β and the window size Δ) and/or support changing the actual solver (e.g., switching to dynamic programming), if needed, to adapt our proposal to changing operating contexts and performance issues that may arise on industry-scale applications.

References

Aiello M, Piontek T, Haghshenas K (2024) Carbon emission-aware job scheduling for Kubernetes deployments. J Supercomput 80. https://doi.org/10.1007/s11227-023-05506-7

Angiolini M, Forti S, Soldani J, et al (2026) Engineering carbon-aware software services. Clust Comp 29(7):439. https://doi.org/10.1007/s10586-026-06212-8

Belkhir L, Elmeligi A (2018) Assessing ICT global emissions footprint: Trends to 2040 & recommendations. J Clean Prod 177:448–463. https://doi.org/10.1016/j.jclepro.2017.12.239

Bhatt H, Biswas S, Rakhunathan S, et al (2025) Harmone: A self-adaptive approach to architecting sustainable MLOps. In: Software Architecture: 19th European Conference. Springer-Verlag, Berlin, Heidelberg, p 38–55, https://doi.org/10.1007/978-3-032-02138-0_3

Broadcom (2026) Rabbitmq. https://www.rabbitmq.com

Bunse C, Höpfner H (2008) Resource substitution with components - optimizing energy consumption. In: Proceedings of the Third International Conference on Software and Data Technologies - Volume 3: ICSOFT, INSTICC. SciTePress, pp 28–35, https://doi.org/10.5220/0001879000280035

Bunse C, Höpfner H, Roychoudhury S, et al (2009) Choosing the "best" sorting algorithm for optimal energy consumption. In: Proceedings of the 4th International Conference on Software and Data Technologies - Volume 2: ICSOFT, INSTICC. SciTePress, pp 199–206, https://doi.org/10.5220/0002245401990206

Couroussé D, Quéva C, Charles HP (2016) Approximate Computing with Runtime Code Generation on Resource-Constrained Embedded Devices. In: 2nd Workshop On Approximate Computing (WAPCO 2016), Vienna, Austria, URL https://cea.hal.science/cea-01296569

Danushi O, Forti S, Soldani J (2025) Carbon-efficient software design and development: A systematic literature review. ACM Comput Surv 57(10):267. https://doi.org/10.1145/3728638

Danushi O, Soldani J, Forti S, et al (2026) Time to go green: Carbon-aware request scheduling in software services. In: 16th International Conference on Cloud Computing and Services Science (CLOSER 2026). SciTePress, Benidorm, Spain, https://doi.org/10.5220/0014725200004039

De Oliveira FGA, Ledoux T (2010) Self-optimisation of the energy footprint in service-oriented architectures. In: Proceedings of the 1st Workshop on Green Computing. ACM, GCM '10, p 4–9, https://doi.org/10.1145/1925013.1925014

Gazza S, Amadini R, Brogi A, et al (2026) A constraint-based approach to optimise qos- and energy-aware cloud-edge application deployments. ACM Trans Internet Technol 26(1). https://doi.org/10.1145/3757061

Giussani S (2026) Architecting carbon-aware software-as-a-service. In: Software Architecture. ECSA 2025 Tracks and Workshops. Springer Nature Switzerland, pp 70–77, https://doi.org/10.1007/978-3-032-04403-7_8

Green Software Foundation (2024) Software carbon intensity (SCI) specification. https://sci.greensoftware.foundation

Green Software Foundation (2025) Carbon-aware SDK. GitHub, https://github.com/Green-Software-Foundation/carbon-aware-sdk

Gsteiger VU, Long PHD, Sun YJ, et al (2024) Caribou: Fine-grained geospatial shifting of serverless applications for sustainability. In: Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. ACM, SOSP '24, p 403–420, https://doi.org/10.1145/3694715.3695954

Hasan MS, de Oliveira FGA, Ledoux T, et al (2016) Enabling green energy awareness in interactive cloud application. In: IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp 414–422, https://doi.org/10.1109/CloudCom.2016.0071

Hasan MS, de Oliveira FGA, Ledoux T, et al (2017) Investigating energy consumption and performance trade-off for interactive cloud application. IEEE Trans Sustain Comput 2(2):113–126. https://doi.org/10.1109/TSUSC.2017.2714959

Hugging Face (2026) The AI community building the future. https://huggingface.co

Kavanagh R, Djemame K (2018) Energy-aware self-adaptive middleware for heterogeneous parallel architectures. In: 2018 5th International Symposium on Innovation in Information and Communication Technology (ISIICT), pp 1–8, https://doi.org/10.1109/ISIICT.2018.8613291

Malmodin J, Lövehagen N, Bergmark P, et al (2024) ICT sector electricity consumption and greenhouse gas emissions – 2020 outcome. Telecommunications Policy 48(3):102701. https://doi.org/10.1016/j.telpol.2023.102701

Merity S, Xiong C, Bradbury J, et al (2016) Pointer sentinel mixture models. CoRR abs/1609.07843. URL http://arxiv.org/abs/1609.07843, 1609.07843

Mingay S, Pamlin D (2008) Assessment of global low-carbon and environmental leadership in the ICT sector. Vol. ID G00161337

Mittal S (2016) A survey of techniques for approximate computing. ACM Comput Surv 48(4). https://doi.org/10.1145/2893356

Radovanovic A (2020) Our data centers now work harder when the sun shines and wind blows. Google Blog, https://blog.google/innovation-and-ai/infrastructure-and-cloud/global-network/data-centers-work-harder-sun-shines-wind-blows/

Souza A, Jasoria S, Chakrabarty B, et al (2023) Casper: Carbon-aware scheduling and provisioning for distributed web services. In: Proceedings of the 14th International Green and Sustainable Computing Conference. ACM, IGSC '23, p 67–73, https://doi.org/10.1145/3634769.3634812

Wohlin C, Runeson P, Höst M, et al (2000) Experimentation in software engineering: an introduction. Kluwer Academic Publishers, Norwell, MA, USA

Xia Y, Aragón-Jurado JM, Thawonmas R (2025) Green by design: Energy-guided reranking of LLM-generated programs. In: Open Conference of AI Agents for Science 2025, URL https://openreview.net/forum?id=BxscqmB9Rs

Xu M, Toosi AN, Buyya R (2021) A self-adaptive approach for managing applications and harnessing renewable energy for sustainable cloud computing. IEEE Trans Sustain Comput 6(4):544–558. https://doi.org/10.1109/TSUSC.2020.3014943