Bridging the Sustainability Gap in Serverless through Observability and Carbon-Aware Pricing

Changyuan Lin
University of British Columbia
chlin@ece.ubc.ca

Mohammad Shahrad
University of British Columbia
mshahrad@ece.ubc.ca

ABSTRACT

Serverless computing has become a mainstream cloud computing paradigm due to its high scalability, ease of server management, and cost-effectiveness. With cloud data centers' carbon footprint rising sharply, understanding and minimizing the carbon impact of serverless functions becomes crucial. The unique characteristics of serverless functions, such as event-driven invocation, pay-as-you-go billing model, short execution duration, ephemeral runtime, and opaque underlying infrastructure, pose challenges in effective carbon metering. In this paper, we argue that the current carbon estimation methodologies should be expanded for more accurate carbon accounting in serverless settings, and propose a usage and allocation-based carbon model that aligns with the context of serverless computing. We also articulate how current serverless systems and billing models do not make it financially attractive to prioritize sustainability for a broad class of users and developers. To solve this, we propose a new carbon-aware pricing model and evaluate its ability to incentivize sustainable practices for developers through better alignment of billing and carbon efficiency.

KEYWORDS

serverless computing, sustainability, carbon metering

1 INTRODUCTION

An increasing portion of greenhouse gas emissions in the Information and Communication Technology (ICT) sector is attributable to the data centers hosting cloud services and resource-intensive workloads. The carbon footprint of cloud data centers keeps increasing fast and is estimated to account for over one-third of ICT carbon emissions by 2030 [38, 57, 62]. Among the various cloud computing services, serverless computing has grown to be a popular paradigm due to its performance and cost benefits and is anticipated to dominate the future of cloud computing [4, 56]. Serverless applications can have a significant carbon footprint depending on configurations and underlying infrastructure [41, 64, 65]. Also, the increasing popularity of artificial intelligence and serverless model serving will generally increase the carbon footprint of serverless further [22, 44, 52]. Therefore, tackling the carbon footprint of serverless functions is a critical step in addressing the sustainability challenge in current cloud systems. This paper aims to reveal the gaps in the carbon observability of serverless functions and propose solutions to bridge the sustainability gap in serverless computing through enhanced carbon observability and carbon-aware pricing.

Currently, serverless developers do not have access to meaningful carbon metrics due to the absence of fine-grained observability frameworks. Tools available on public cloud platforms, such as Customer Carbon Footprint Tool [8] and Emissions Impact Dashboard [7], only provide coarse-grained carbon metrics (e.g., per-region and per-service) with limited observability. While recent studies and open-source projects have proposed several models and tools to capture the carbon footprint of software systems [18, 36, 46], they are not effective in capturing and modeling the carbon emissions of serverless functions due to not considering unique serverless characteristics such as event-driven invocation, pay-as-you-go billing model, short execution duration, ephemeral runtime sandbox, and opaque underlying infrastructure. Such a gap hinders fine-grained carbon emission analysis on per-function and per-request levels.

Offering meaningful incentives is key to driving sustainable practices across all sectors, including in cloud systems. The primary incentive for providers is to reduce the carbon footprint of their cloud services in response to the sustainability target and regulations. Cloud providers, such as AWS and Microsoft Azure, have already launched carbon reduction initiatives in recent years [6, 9]. Similarly, environmentally conscious organizations are already on board as long as they are provided with meaningful carbon observability means [60]. However, current serverless systems and billing

HotCarbon'24, July 9, 2024, Santa Cruz, CA

models do not make it financially attractive to prioritize sustainability for a broad class of users and developers, whose concerns are usually centered around performance. In this context, this work's primary contributions include:

(1) We identify unsustainable practices arising from the misalignment between developer goals and sustainability objectives in today's serverless systems.
(2) To bridge the sustainability gap, we propose a carbon model tailored for serverless and propose a feasible system design for collecting carbon-related metrics and providing enhanced carbon observability.
(3) We evaluate our carbon model and the potential of carbon-aware pricing through case studies, demonstrating its effectiveness in incentivizing sustainable practices.

2 BACKGROUND AND THE STATUS QUO

Serverless resource allocation: Serverless functions usually leverage lightweight virtualization and resource isolation techniques, such as micro virtual machines (VMs) and *cgroups*-based containers (or Kubernetes pods), to achieve minimal performance overhead and enhanced scalability. Developers typically only need to provide the function code or container images. Serverless platforms handle operational tasks, such as initialization, scaling, request routing, prewarming or keep-alive policies, and accounting. Serverless platforms, including managed services like AWS Lambda [31] and open-source solutions like Knative [21] and Apache OpenWhisk [25], have become widely adopted.

Observability in Serverless: Observability is the ability to reason about the system status based on logs, traces, and metrics. Many open-source projects like *Grafana* [72] have focused on serverless observability for metrics/log collection and analysis. While they are not directly designed for carbon observability, the metrics/logs and data pipelines can be used to analyze carbon emissions. Several studies and tools provide energy monitoring solutions for containers, including *Kepler* [36], *Scaphandre* [18], *SmartWatts* [46], and *PowerAPI* [40]. They retrieve power metrics from Running Average Power Limit (RAPL) interfaces and use CPU and memory usage metrics, along with power models, to estimate the energy consumption of each process or container. The focus of these tools on the energy consumption of long-running applications makes them unfit for unique serverless patterns, such as the short execution duration, event-driven architecture, ephemeral container, and keep-alive policies for cold start mitigation. They usually have a sampling rate of up to 1 Hz, while over 50% of functions have a duration within a second, let alone the function's ephemeral subprocesses [54, 67]. Some works [18, 42] perform CPU accounting based on jiffies that mismatch the function billing granularity (1 ms). Thus, they provide only coarse-grained energy metrics that are not effective in tracing the carbon footprint of individual function requests. Lack of fine-grained and low-overhead energy/carbon metering leads to inaccurate carbon accounting and limits usability for developers.

Power consumption and carbon footprint models: The power monitoring solutions mentioned above [36, 40, 46] and many existing power models [50, 51, 66] typically leverage a model to calculate the share of power that a monitored application $\mathcal{A}$ should bear based on its resource usage, denoted as $P^{\mathcal{A}}$. It can be generally formulated as

$$P^{\mathcal{A}} = \sum_{r \in R} \left[P^r_{sta} \times AT_{sta}\,(r, \mathcal{A}) + P^r_{dyn} \times AT_{dyn}\,(r, \mathcal{A}) \right] \quad (1)$$

where P^r_{sta} and P^r_{dyn} are the static (idle) and dynamic power of the subsystem offering resource r, and AT defines the fraction of power attributed to the application (e.g., ratio based on usage). While this power model has been proven to be accurate [50, 54], solely using power consumption models fails to assess the holistic carbon footprint of serverless functions. Embodied carbon and unique patterns in serverless are important considerations [49, 58]. Existing long-running metrics collection methods are often ill-suited for serverless functions, again due to not being event-driven and due to the mismatch of sampling rate and metering granularity.

Serverless billing model: One of the most attractive features of serverless for developers is its pay-as-you-go billing model with a billing granularity of as small as milliseconds, minimizing idle costs. In its most general form, the cost is determined by function execution duration d, the allocated resource size ALC (e.g., memory size and ephemeral block storage size) for serving the function request, the amount of actual resource usage USG of the request (e.g., data transfer amount), and a fixed per-request cost c_0. Formally, we can define the cost C of a request q for serverless function f as:

$$C^q_f = \sum_{r \in R_A} ALC\,(f, q, r) \times d \times \mathbf{c}_r + \sum_{r \in R_U} USG\,(q, r) \times \mathbf{c}'_r + c_0 \quad (2)$$

Here, R_A represents the set of allocation-billed resources, $\mathbf{c}_r$ is the cost per unit resource size and duration (e.g., GB-second), R_U is the set of usage-billed resources, and $\mathbf{c}'_r$ is the cost per unit of resource usage. For instance, AWS Lambda charges $\$1.66667 \times 10^{-5}$ for every GB-second of the memory allocation (with proportional CPU allocation), $\$3.09 \times 10^{-8}$ for every GB-second of the additional block storage allocation beyond 512 MB, and a fixed per-request fee of $\$2 \times 10^{-7}$ [32].

While pay-as-you-go billing enables cost-effectiveness, **the current serverless billing model fails to reflect a function's environmental impact in its cost**. For instance, consider a CPU-bound function and another with a long, non-busy wait for remote database transactions to complete. If both functions request identical resources (i.e., memory sizes and #vCPUs) and happen to have similar durations, they would cost the same. When it comes to carbon, however,

Serverless Platform	Billable Time	Init	Exec	KA (Idle)	Shut	Reference
AWS Lambda [31]	Execution Time	✗	✓	✗* / ✓[1]	✗* / ✓[4]	[23, 32]
Google Cloud Functions [10]	Turnaround Time	✓	✓	✗* / ✓[2]	✗	[26, 28]
Google Cloud Run [11] (Default Allocation Mode)	Turnaround Time	✓	✓	✗* / ✓[2]	✓	[15, 29]
Google Cloud Run [11] (Always Allocated Mode)	Instance Time	✓	✓	✓	✓	[15, 29]
Azure Functions [5] (Consumption Tier)	Execution Time	✗	✓	✗	✗	[27]
IBM Cloud Code Engine [19] (Function Workloads)	Turnaround Time	✓	✓	✗* / ✓[3]	✗	[30, 35]
IBM Cloud Code Engine [19] (Application Workloads)	Instance Time	✓	✓	✓[2,3]	✓	[14, 30]

*Not billed under the default configuration. Users may customize the keep-alive (KA) policy by configuring provisioned concurrency[1], minimum instances[2], or scale-down delay[3] and pay for idle resources. [4]Billed for functions with AWS Lambda Extensions.

Table 1: The notion of billable time varies across different serverless platforms.

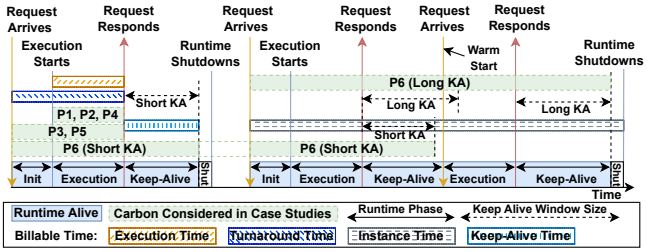

Figure 1: The lifecycle stages of the function runtime sandbox (e.g., container) and the related carbon-based cost discussed in case studies presented in §3 and §5.

the latter would have lower emissions. This simple example highlights a gap in the serverless billing model, which fails to align the cost with the carbon emissions of functions.

An important factor influencing the cost of serverless functions is which lifecycle stage is billed. This differs across serverless platforms (Table 1). Besides execution duration, it may involve various lifecycle stages (Figure 1), such as initialization (cold start), shutdown, and keep-alive (KA) time for cold start mitigation. Providers usually use a similar pricing model as shown in Eq. (2), but with varying definitions for billable time. Unlike AWS Lambda's billing based solely on function execution, Google Cloud Functions (GCF) charges by default for the entire turnaround time, including resource initialization after request arrival. In some other cases, a cloud provider may use instance time for billing purposes, which is the lifecycle duration of a function runtime instance (e.g., pod), and may include the instance initialization time after the request arrival, execution duration, keep-alive time, and shutdown time (i.e., handling *SIGTERM*). Also, some platforms may have minimum cutoff and round-up policies for billable time; e.g., Azure Functions has a minimum billed duration of 100 ms [27], or GCF bills in 100 ms increments by rounding up the duration to the nearest increment [28].

3 EXAMPLES OF UNSUSTAINABLE PRACTICES IN SERVERLESS

As mentioned in §1, reasonable incentives can accelerate sustainable practice adoption. Current serverless systems and billing models often fail to make sustainability financially attractive for a broad class of users and developers with concerns centered around performance. We dedicate this section to identifying specific friction points between developer goals and sustainability goals.

P1: Resource wastage due to inflexible allocation: Serverless platforms can have inflexible resource allocation options; e.g., AWS Lambda allocates CPU in proportion to the memory size with a ratio of 0.000565 vCPUs/MB [13]. Inflexible allocation forces developers to waste some resources to feed the bottleneck resource [39]. For instance, they may allocate more memory than necessary to achieve sufficient CPU power for optimal performance and/or cost efficiency [63]. This over-allocation results in unnecessary emissions.

P2: Ignored memory access patterns: The last-level cache misses can incur excessive memory accesses and carbon emissions for the memory subsystem [59, 61] and slow down function execution. The carbon impact (not captured in billing) of such misses is not necessarily proportional to their performance impact (captured in billing).

P3: Minimum billing cutoff and coarse-grained rounding: Serverless platforms may have a minimum billing period for function execution duration. For example, Azure Functions enforces a minimum duration of 100 ms in billing, while around 25% of functions have a response time less than this minimum billing period on average [27, 67]. GCF rounds up function execution duration to the nearest multiple of 100 ms [28]. Such billing policies discourage efficiency improvements for users who are not focused on latency gains.

P4: Default configuration combined with lack of carbon observability: Many studies have discussed that the default function configurations are generally not cost and performance optimal [13, 45, 63]. The same applies to carbon optimality, as each function has specific resource requirements and usage patterns. While cost and performance are readily logged for a developer to optimize for, carbon is not.

P5: Large code base: A large function code base with many dependencies generally requires a longer time to initialize (i.e., cold start), during which resources are intensively used, thereby causing higher carbon emissions. As cold start latency may not be billed by the provider [23] and can be mitigated by keep-alive and pre-warming policies [47, 67], minimal software packages lack appeal for developers.

P6: Suboptimal lifetime management policy: Cloud providers may have a fixed keep-alive policy for function containers with varied allocated resources [2]. Carbon emissions of idle containers and cold starts can serve as incentives for cloud providers and users to devise optimal function container lifecycle management policies.

The responsibility for these unsustainable practices can fall on providers, users, or both. It is the provider's responsibility to offer flexible resource allocation options (P1) and fine-grained billing (P3). Since users have full control over their source code and configurations, the onus is on them in cases P2, P4, and P5. Providers bear primary responsibility for P6: they can use cold start mitigation strategies (e.g., pre-warming [67]) to strike a balance between the carbon emissions of initialization and idle time. Meanwhile, users also play a role for P6 due to their control over the code base size and their ability to influence the keep-alive policy through configuration (e.g., scale-down delay).

4 PROPOSED SOLUTION

The sustainability issues presented in §3 stem from the limited carbon observability and the lack of a carbon-aware pricing model in current serverless computing systems. We propose a solution to bridge this sustainability gap, aimed at providing reasonable incentives for stakeholders and stimulating effective sustainability actions in serverless computing. Our proposal is centered around identifying critical metrics for function carbon accounting (§4.1), developing a new carbon model for serverless functions (§4.2), and incorporating relevant metrics and the carbon model in a carbon-aware pricing model that better aligns cost and carbon (§4.4).

4.1 System Metrics Affecting Emissions

In serverless computing, a container (or K8s pod) is usually the basic deployment unit, with resources allocated on a host server. The metrics to measure the resource utilization of function containers can be categorized into *allocation-based* metrics and *usage-based* metrics, both of which are critical for modeling the carbon footprint of serverless functions.

Allocation-based metrics reflect the amount of resources allocated to functions. From the scheduling perspective, each container has a resource request vector defining the amount for each type of required resource, such as the number or fraction of vCPU cores, memory size, block storage size, reserved network capacity, and GPU memory size. The container scheduler (e.g., *kube-scheduler*) places the function container based on this vector and the available capacity on suitable servers. Resources are dedicated once allocated and cannot be used by other functions regardless of actual usage. Therefore, allocation-based metrics generally contribute to the static power and the embodied carbon of the function's host server. Reliable metrics are needed to map the resource request vector to carbon emissions.

Usage-based metrics capture the actual utilization of resources by functions, which are provided by hardware performance counters or kernel features (e.g., *Cgroups*). The actual resource usage typically determines dynamic power and parts of embodied carbon, depending on the nature of the device lifecycle. For example, providers may define SSD replacement policies based on the wear from writing operations. The function's actual I/O usage determines the share of the storage subsystem embodied carbon it should bear in this case. Relevant usage-based metrics for power estimation can vary depending on the hardware platform and configurations. Key usage-based metrics in the latest literature and tools [18, 36, 46, 50, 66] include CPU time, memory usage, CPU cycles, instruction count, #cache misses, and #I/O operations and amount for the network and block storage.

4.2 Carbon Model Formulation

The carbon footprint of serverless functions consists of operational and embodied carbon. Operational carbon primarily arises from power consumption and is influenced by the data center power usage effectiveness (PUE) and the carbon intensity (CI) of the used energy. It is crucial that the carbon model for serverless functions is meaningfully defined on a per-request basis. A per-request carbon model aligns with serverless's pay-as-you-go billing and event-driven nature, and effectively captures the variable emissions of input-sensitive functions. For a function f with a set of allocated resources R placed on server s in data center l, the carbon footprint CB for serving the request (a.k.a. invocation) q is:

$$CB_f^q = CO_f^q + CE_f^q \tag{3}$$

CO_f^q and CE_f^q are the operational and embodied carbon attributed to the function request, with former calculated as

$$CO_f^q = \sum_{r \in R} \int_{\tau}^{\tau+d} P_f^q(r,t) \times PUE(l,t) \times CI(l,t)\, dt \tag{4}$$

where τ is the execution start time, d is the duration. The power attributed to function resource usage over time is

$$P_f^q(r,t) = P_{sta}^r \times \frac{ALC(f,q,r)}{CAP(s,r)} + P_{dyn}^r(t) \times \frac{USG(q,r,t)}{USG(s,r,t)} \tag{5}$$

where $ALC(f,q,r)$ is the allocated size of resource r to f for serving request q, CAP represents the total capacity of resource r on the host, USG defines the usage amount of a type of resource by the function request or the server over time, and P_{sta}^r and P_{dyn}^r are the static and dynamic power of the subsystem on the host server for resource r.

We calculate the embodied carbon of a request as

$$CE_f^q = \sum_{r \in R} CE_r \times \frac{ALC(f,q,r)}{CAP(s,r)} \times d \tag{6}$$

where CE_r is the embodied carbon of the hardware associated with the resource r in per resource unit and duration. For embodied carbon defined on a per-usage basis, the attribution based on the multiplication of allocation ratio with duration should be replaced with the amount of actual resource usage.

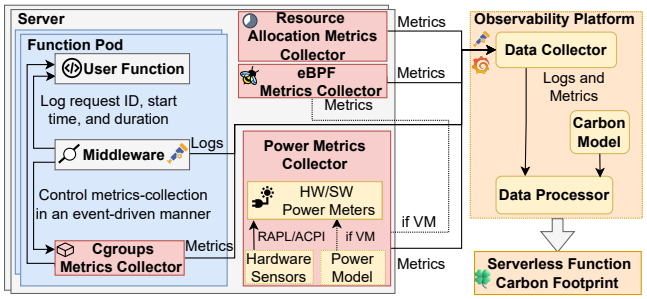

Figure 2: The architecture of the proposed metrics collection system and carbon observability framework.

The same formulation is applicable to model the carbon footprint of the other function container life stages, such as the initialization and the keep-alive phases. For instance, the function is kept alive for a certain period, followed by a warm start when it is invoked again within the keep-alive window. Since few resources are used during the keep-alive period (mainly memory), the carbon footprint at this phase is dominated by the static power and the embodied carbon.

4.3 Metrics Collection

An efficient system to collect carbon-related metrics is essential to assess the carbon footprint of serverless functions accurately. Figure 2 demonstrates the proposed system architecture. The required components in the metrics collection and observability framework are primarily based on open-source projects and cloud-native tools, such as *eBPF* [16], *OpenTelemetry* [24], and *Grafana*, ensuring system feasibility. For accurate per-request carbon footprints, the architecture targets serverless platforms with non-interleaving concurrency models. This is similar to the non-interleaving concurrency models used in AWS Lambda [34] and the default configuration of GCF [12]. This setup allows for differentiating carbon emission sources for each request and simplifies event-driven metrics collection.

For power metrics, each server/node runs a collector that fetches real-time power consumption metrics from the RAPL, ACPI, or Redfish interface at regular intervals. For VM-based nodes, energy metrics can be either passed through by the hypervisor or derived from learning-based models to estimate the static and dynamic power based on metrics [36]. The *eBPF* metrics collector periodically collects low-level, usage-based metrics from hardware performance counters for processes in each runtime sandbox, such as CPU cycles and cache misses. It may also gather NUMA-aware metrics (e.g., CPU time on each socket) if containers can run on different sockets [50]. The *Cgroups* collector runs as a sidecar controlled by function middleware (HTTP handler) in an event-driven manner. It gathers usage metrics like CPU time and I/O volume from *Cgroupfs* at function execution start, end, and periodically throughout. The resource allocation metrics collector provides allocation-based metrics using the function runtime orchestration framework (e.g., Kubernetes).

Besides metrics, prior work has shown that request traces are essential for providing carbon footprints for applications at a fine-grained request level [37]. We leverage the function middleware and distributed tracing techniques (e.g., OpenTelemetry) to keep a log of function execution, such as function invocation request ID, execution start time, and execution duration. With function execution logs, the usage-based and allocation-based metrics obtained by various collectors with different sampling rates can be linked to each individual function request, thereby enabling carbon modeling on a per-request basis. The metrics and logs are pushed to the observability platform (e.g., Grafana) and used to calculate the carbon emission of individual function requests with the carbon model discussed in §4.2.

4.4 Carbon-Aware Pricing

Leveraging pricing to enhance the efficiency of cloud systems is not new [68]. Recent studies have highlighted the need for pricing models that can effectively translates carbon emissions into cost [48, 69]. However, to the best of our knowledge, carbon-aware pricing in the context of serverless to incentivize adoption of sustainable practices has not been proposed before. Possible approaches may include 1) a straightforward pricing model that charges users per unit of carbon emission, 2) a tiered pricing model with different pricing tiers based on emission levels, and 3) a mixed pricing model that combines the current billing model (Eq. (2)) at a reduced rate and a carbon tax surcharge based on emissions. Instead of proposing an exact carbon-based pricing model, we argue that serverless providers need to explore carbon-aware pricing policies that fit their specific operational and financial contexts (e.g., cloud infrastructure and revenue models) and sustainability objectives. In practice, functions may be billed on historical average values of carbon elements (e.g., PUE and CI) to ensure pricing consistency or dynamic values that reflect the real-time carbon to incentivize temporal and spatial workload shifting [70]. In §5, we show that even a basic proportional pricing model has the potential to disincentivize unsustainable practices in serverless.

5 EVALUATION THROUGH CASE STUDIES

To validate the incentives provided by the carbon model and carbon-aware billing, we conduct case studies to estimate the cost of serverless functions with unsustainable practices discussed in §3 and the corresponding sustainable practices. We use a learning-based power model from the *Kepler* Model

	Unsustainable Practice	Sustainable Practice	Current Cost Ratio* Sustainable/Unsus	Proposed Carbon Cost Ratio Sustainable/Unsus	Notes on Specific Scenario**
P1	Configure 3,538 MB to get 2 vCPUs	Configure 512 MB of memory and 2 vCPUs	N/A[1] (Rigid Configuration)	0.980	Workload CPU time 200 ms
P2	Ignore memory access patterns	Improve cache behaviors to reduce cache misses by 20%; CPU time is reduced by 1 ms	0.989[1]	0.863	Workload CPU time 20 ms; 0.25 vCPUs, Memory 443 MB
P3	Minimum billing cutoff	Improve function performance by reducing CPU time by 2 ms	1.000[2]	0.821	Total CPU time 25 ms; 0.167 vCPUs, Memory 256 MB
P4	Default configuration	Increase memory size from 128 MB to 512 MB	0.970[1]	0.305	Workload CPU time 20 ms
P5	Large code base	Shrink code base and reduce the initialization CPU time by 20%; The original initialization CPU time is 30 ms	1.000[2]	0.954	Workload CPU time is 100 ms; 0.167 vCPUs, Memory 256 MB
P6	Additional cold starts due to fixed container lifetime management policy not suitable for the invocation pattern	Adjust the 10-second keep-alive window size to 11 seconds to eliminate cold starts	1.042[3]	0.962	Workload CPU time 1,000 ms; Initialization CPU time 100 ms; 0.25 vCPUs, Memory 1 GB; 2 reqs with 15 s inter-arrival time

*Billing model used: AWS Lambda[1], Google Cloud Functions[2], and IBM Cloud Code Engine (function workloads) with keep-alive[3]. **CPU time is the actual amount of time taken by a CPU core to execute the function, which can be shorter than the wall-clock execution duration due to fractional CPU allocations. For example, the execution duration of a function that requires 20 ms of CPU time for the workload is around 65 ms with 0.25 vCPUs and a 20 ms *cgroup* period.

Table 2: The ratio of costs of serverless functions with sustainable and unsustainable practices under the current serverless billing model and the carbon-aware billing policy.

DB [33] to estimate the power usage of CPU and DRAM components in AWS bare metal instances (i.e., *i3.metal*) [20]. The model is trained on power and usage-based metrics collected from RAPL and *eBPF* with mean absolute percentage errors of 1.0% and 4.1% for static and dynamic power estimations. For the embodied carbon, we refer to the average values for SSDs, CPUs, and memory reported in recent studies [1, 71]. We adopt the embodied carbon of 652.78 gCO2 for a CPU core and 1.39 and 0.11 KgCO2/GB for the memory and SSDs and a standard operational lifespan of five years. We use a PUE of 1.11, the average of the 1.07-1.15 range reported for AWS datacenters [3], and CI of 42 gCO2/kWh for the datacenter [17]. Besides, we adopted an average server CPU utilization of 45% [43], an average last-level cache miss rate of 5 per 1,000 cycles [53], and an average cycle per instruction of 1.58 [55].

We aim to compare a pricing model that charges per unit of carbon emission to existing billing models of public serverless platforms. To do so, we simulate functions with specific resource allocation (i.e., vCPUs and memory) and the CPU time required to execute the function for each scenario. Table 2 presents the comparison between the ratios of costs of serverless functions with sustainable practices discussed in §3 over those with unsustainable practices, under both current billing models (Eq. (2)) and the carbon-aware billing policy. Figure 1 illustrates the carbon emissions for different runtime stages considered in each case study.

As shown in Table 2, the cost ratios under the carbon-aware billing model are consistently lower than those based on current billing models (if applicable). For instance, in P2, improving cache behaviors translates into a 1.1% cost reduction under the current billing model. In contrast, the cost reduction is 13.7% with the carbon-aware billing. This generally demonstrates that the carbon-aware billing policy is more sensitive to carbon efficiency improvements. Also, carbon-aware pricing can effectively reflect carbon reduction and guide relevant optimization decisions for providers and users. In P6, the carbon-based billing incentivizes addressing suboptimal keep-alive configurations that can cause more cold starts. Conversely, the current billing model may discourage this optimization decision. Such results demonstrate the potentials of the proposed carbon model and the carbon-aware pricing policy in offering incentives for the adoption of sustainable practices in serverless computing.

6 CONCLUSION AND VISION

The unique operational and architectural characteristics of serverless systems present challenges to achieving carbon observability and, ultimately, sustainability. Broad adoption of sustainable practices by users requires incentives, which are currently constrained by the limited carbon observability and the carbon-agnostic billing model.

In this paper, we proposed carbon models and a metrics collection system tailored for sustainable serverless computing. Our fine-grained carbon observability framework aims to deliver per-request carbon footprint metrics with minimal overhead and complexity, seamlessly integrating with existing cloud-native serverless systems. We also argued that it is valuable to design carbon-aware pricing models that translate carbon into cost with pricing consistency and incentivize sustainable practices for both providers and developers. By identifying and simulating the impact of carbon emission on specific unsustainable user/provider behaviors, we showed how enhanced carbon observability and carbon-aware pricing bridge the sustainability gap in future serverless systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and members of the UBC CIRRUS Lab for their valuable feedback. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grants RGPIN-2021-03714 and DGECR-2021-00462.

REFERENCES

[1] Building an AWS EC2 carbon emissions dataset by benjamin davy. https://medium.com/teads-engineering/building-an-aws-ec2-carbon-emissions-dataset-3f0fd76c98ac, 2021. [Online; accessed May-5-2024].

[2] Comparison of cold starts in serverless functions across AWS, Azure, and GCP. https://mikhail.io/serverless/coldstarts/big3/, 2021. [Online; accessed May-5-2024].

[3] Four trends driving global utility digitization. https://aws.amazon.com/blogs/industries/four-trends-driving-global-utility-digitization/, 2022. [Online; accessed May-5-2024].

[4] The state of serverless | datadog. https://www.datadoghq.com/state-of-serverless/, 2023. [Online; accessed May-3-2024].

[5] Azure Functions – serverless functions in computing | Microsoft Azure. https://azure.microsoft.com/en-ca/products/functions, 2024. [Online; accessed June-24-2024].

[6] Azure sustainability—sustainable technologies | Microsoft Azure. https://azure.microsoft.com/en-us/explore/global-infrastructure/sustainability, 2024. [Online; accessed May-4-2024].

[7] Calculating my carbon footprint | Microsoft sustainability. https://www.microsoft.com/en-ca/sustainability/emissions-impact-dashboard, 2024. [Online; accessed May-4-2024].

[8] Carbon footprint reporting – customer carbon footprint tool – amazon web services. https://aws.amazon.com/aws-cost-management/aws-customer-carbon-footprint-tool/, 2024. [Online; accessed May-4-2024].

[9] The cloud - amazon sustainability. https://sustainability.aboutamazon.com/products-services/the-cloud, 2024. [Online; accessed May-4-2024].

[10] Cloud functions | google cloud. https://cloud.google.com/functions, 2024. [Online; accessed June-24-2024].

[11] Cloud run | google cloud. https://cloud.google.com/run, 2024. [Online; accessed June-24-2024].

[12] Concurrency | cloud functions documentation | google cloud. https://cloud.google.com/functions/docs/configuring/concurrency, 2024. [Online; accessed June-21-2024].

[13] Configure Lambda function memory - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html, 2024. [Online; accessed May-5-2024].

[14] Configuring application scaling | ibm cloud docs. https://cloud.ibm.com/docs/codeengine?topic=codeengine-app-scale, 2024. [Online; accessed June-26-2024].

[15] Cpu allocation (services) | cloud run documentation | google cloud. https://cloud.google.com/run/docs/configuring/cpu-allocation, 2024. [Online; accessed June-24-2024].

[16] eBPF - introduction, tutorials & community resources. https://ebpf.io/, 2024. [Online; accessed May-4-2024].

[17] Electricity maps | reduce carbon emissions with actionable electricity data. https://www.electricitymaps.com/, 2024. [Online; accessed May-5-2024].

[18] hubblo-org/scaphandre: Energy consumption metrology agent. let "scaph" dive and bring back the metrics that will help you make your systems and applications more sustainable ! https://github.com/hubblo-org/scaphandre, 2024. [Online; accessed May-5-2024].

[19] Ibm cloud code engine. https://www.ibm.com/products/code-engine, 2024. [Online; accessed June-24-2024].

[20] kepler-model-db/models/v0.7/ec2 at main · sustainable-computing-io/kepler-model-db. https://github.com/sustainable-computing-io/kepler-model-db/tree/main/models/v0.7/ec2, 2024. [Online; accessed May-5-2024].

[21] Knative. https://knative.dev/, 2024. [Online; accessed June-18-2024].

[22] kserve/kserve: Standardized serverless ml inference platform on kubernetes. https://github.com/kserve/kserve, 2024. [Online; accessed May-5-2024].

[23] Lambda execution environments - aws lambda. https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html, 2024. [Online; accessed May-5-2024].

[24] Opentelemetry. https://opentelemetry.io/, 2024. [Online; accessed June-16-2024].

[25] openwhisk. https://openwhisk.apache.org/, 2024. [Online; accessed June-18-2024].

[26] Optimize cost: Compute, containers, and serverless | cloud architecture center | google cloud. https://cloud.google.com/architecture/framework/cost-optimization/compute#functions, 2024. [Online; accessed June-24-2024].

[27] Pricing - functions | Microsoft Azure. https://azure.microsoft.com/en-ca/pricing/details/functions/, 2024. [Online; accessed May-4-2024].

[28] Pricing | cloud functions | google cloud. https://cloud.google.com/functions/pricing, 2024. [Online; accessed June-20-2024].

[29] Pricing | cloud run | google cloud. https://cloud.google.com/run/pricing, 2024. [Online; accessed June-24-2024].

[30] Pricing for code engine | ibm cloud docs. https://cloud.ibm.com/docs/codeengine?topic=codeengine-pricing, 2024. [Online; accessed June-24-2024].

[31] Serverless computing - AWS Lambda - Amazon Web Services. https://aws.amazon.com/lambda/, 2024. [Online; accessed June-24-2024].

[32] Serverless computing – AWS Lambda pricing – Amazon Web Services. https://aws.amazon.com/lambda/pricing/, 2024. [Online; accessed May-6-2024].

[33] sustainable-computing-io/kepler-model-db: Repository containing up-to-date models to be used by the kepler-model-server. https://github.com/sustainable-computing-io/kepler-model-db, 2024. [Online; accessed May-5-2024].

[34] Understanding lambda function scaling - aws lambda. https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html, 2024. [Online; accessed June-18-2024].

[35] Working with functions | ibm cloud docs. https://cloud.ibm.com/docs/codeengine?topic=codeengine-fun-work, 2024. [Online; accessed June-24-2024].

[36] Marcelo Amaral, Huamin Chen, Tatsuhiro Chiba, Rina Nakazawa, Sunyanan Choochotkaew, Eun Kyung Lee, and Tamar Eilam. Kepler: A framework to calculate the energy consumption of containerized applications. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 69–71. IEEE, 2023.

[37] Vaastav Anand, Zhiqiang Xie, Matheus Stolet, Roberta De Viti, Thomas Davidson, Reyhaneh Karimipour, Safya Alzayat, and Jonathan Mace. The odd one out: Energy is not like other metrics. *ACM SIGENERGY Energy Informatics Review*, 3(3):71–77, 2023.

[38] Anders SG Andrae and Tomas Edler. On global electricity usage of communication technology: trends to 2030. *Challenges*, 6(1):117–157, 2015.

[39] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 381–397, New York, NY, USA, 2023. ACM.

[40] Aurélien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Powerapi: A software library to monitor the energy consumed at the process-level. *ERCIM News*, 92:43–44, 2013.

[41] Mohak Chadha, Thandayuthapani Subramanian, Eishi Arima, Michael Gerndt, Martin Schulz, and Osama Abboud. Greencourier: Carbon-aware scheduling for serverless functions. In *Proceedings of the 9th International Workshop on Serverless Computing*, pages 18–23, 2023.

[42] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. Greenscaler: training software energy models with automatic test generation. *Empirical Software Engineering*, 24:1649–1692, 2019.

[43] Li Deng, Yu-Lin Ren, Fei Xu, Heng He, and Chao Li. Resource utilization analysis of Alibaba cloud. In *Intelligent Computing Theories and Application: 14th International Conference, ICIC 2018, Wuhan, China, August 15-18, 2018, Proceedings, Part I 14*, pages 183–194. Springer, 2018.

[44] Payal Dhar. The carbon impact of artificial intelligence. *Nat. Mach. Intell.*, 2(8):423–425, 2020.

[45] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*, pages 248–259, 2021.

[46] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. Smartwatts: Self-calibrating software-defined power meter for containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 479–488. IEEE, 2020.

[47] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[48] Anshul Gandhi, Dongyoon Lee, Zhenhua Liu, Shuai Mu, Erez Zadok, Kanad Ghose, Kartik Gopalan, Yu David Liu, Syed Rafiul Hussain, and Patrick Mcdaniel. Metrics for sustainability in data centers. *ACM SIGENERGY Energy Informatics Review*, 3(3):40–46, 2023.

[49] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. Act: Designing sustainable computer systems with an architectural carbon modeling tool. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 784–799, 2022.

[50] Hongyu Hè, Michal Friedman, and Theodoros Rekatsinas. Energat: Fine-grained energy attribution for multi-tenancy. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, pages 1–8, 2023.

[51] Franz Christian Heinrich, Tom Cornebize, Augustin Degomme, Arnaud Legrand, Alexandra Carpen-Amarie, Sascha Hunold, Anne-Cécile Orgerie, and Martin Quinson. Predicting the energy-consumption of MPI applications at scale using only a single node. In *2017 IEEE international conference on cluster computing (CLUSTER)*, pages 92–102. IEEE, 2017.

[52] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International conference on cloud engineering (IC2E)*, pages 257–262. IEEE, 2018.

[53] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C Steely Jr, and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–162. IEEE, 2010.

[54] Mathilde Jay, Vladimir Ostapenco, Laurent Lefèvre, Denis Trystram, Anne-Cécile Orgerie, and Benjamin Fichel. An experimental comparison of software-based power meters: focus on CPU and GPU. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 106–118. IEEE, 2023.

[55] Congfeng Jiang, Yitao Qiu, Weisong Shi, Zhefeng Ge, Jiwei Wang, Shenglei Chen, Christophe Cérin, Zujie Ren, Guoyao Xu, and Jiangbin Lin. Characterizing co-located workloads in Alibaba cloud datacenters. *IEEE Transactions on Cloud Computing*, 10(4):2381–2397, 2020.

[56] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[57] Nicola Jones et al. How to stop data centres from gobbling up the world's electricity. *Nature*, 561(7722):163–166, 2018.

[58] Sudarsun Kannan and Ulrich Kremer. Towards application centric carbon emission management. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, pages 1–7, 2023.

[59] Bhavani Krishnan, Hrishikesh Amur, Ada Gavrilovska, and Karsten Schwan. VM power metering: feasibility and challenges. *ACM SIGMETRICS Performance Evaluation Review*, 38(3):56–60, 2011.

[60] Daphne Leprince-Ringuet. How clean is cloud computing? new data reveals how green Google's data centers really are. https://www.zdnet.com/article/how-clean-is-cloud-computing-new-data-reveals-how-green-googles-data-centers-really-are/, 2021. Accessed: 2024-04-17.

[61] Jiang Lin, Hongzhong Zheng, Zhichun Zhu, Eugene Gorbatov, Howard David, and Zhao Zhang. Software thermal management of dram memory for multicore systems. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):337–348, 2008.

[62] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[63] Arshia Moghimi, Joe Hattori, Alexander Li, Mehdi Ben Chikha, and Mohammad Shahrad. Parrotfish: Parametric regression for optimizing serverless functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 177–192, 2023.

[64] Panos Patros, Josef Spillner, Alessandro V Papadopoulos, Blesson Varghese, Omer Rana, and Schahram Dustdar. Toward sustainable serverless computing. *IEEE Internet Computing*, 25(6):42–50, 2021.

[65] Alexander Poth, Niklas Schubert, and Andreas Riel. Sustainability efficiency challenges of modern it architectures–a quality model for serverless energy footprint. In *Systems, Software and Services Process Improvement: 27th European Conference, EuroSPI 2020, Düsseldorf, Germany, September 9–11, 2020, Proceedings 27*, pages 289–301. Springer, 2020.

[66] Norbert Schmitt, Lukas Iffländer, André Bauer, and Samuel Kounev. Online power consumption estimation for functions in cloud applications. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 63–72. IEEE, 2019.

[67] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.

[68] Mohammad Shahrad, Cristian Klein, Liang Zheng, Mung Chiang, Erik Elmroth, and David Wentzlaff. Incentivizing self-capping to increase cloud utilization. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 52–65. ACM, 2017.

[69] Prateek Sharma. Challenges and opportunities in sustainable serverless computing. *ACM SIGENERGY Energy Informatics Review*, 3(3):53–58, 2023.

[70] Thanathorn Sukprasert, Abel Souza, Noman Bashir, David Irwin, and Prashant Shenoy. On the limitations of carbon-aware temporal and spatial workload shifting in the cloud. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 924–941, 2024.

[71] Swamit Tannu and Prashant J Nair. The dirty secret of SSDs: Embodied carbon. *ACM SIGENERGY Energy Informatics Review*, 3(3):4–9, 2023.

[72] Harry Thornburg. Grafana: The open observability platform | grafana labs, mar 2001.