EcoTracker: An Application-Level Carbon Scheduling Framework for LLMs

Adomas Bagdonas, Georgi Dimitrov, Kristian Hristov, Stilyan Penchev, Daniel Rachev
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Delft, Netherlands

***Abstract*—The rapid growth of Large Language Models (LLMs) has intensified concerns about the environmental cost of AI inference, particularly for developers who rely on third-party cloud APIs and have no control over the underlying infrastructure. We present EcoTracker, an open-source Python framework published on the Python Package Index (PyPI) for application-level carbon-aware execution of delay-tolerant LLM workloads. EcoTracker combines forecast-guided temporal shifting with optional policy mechanisms such as model downgrading, carbon-budget enforcement, and normalized telemetry collection through a decorator-based interface. In a controlled benchmark using a fixed carbon-intensity trace, EcoTracker was evaluated across 1,392 scheduling scenarios over 29 days. Relative to immediate execution, it reduced total emitted carbon by 9.99% and captured 76.9% of the savings achieved by an oracle policy with perfect hindsight. These results show that meaningful operational emissions reductions can be achieved at the application layer without requiring infrastructure-level control, suggesting that lightweight software mechanisms can complement broader efforts toward more sustainable AI systems.**

I. Introduction

The rapid advancement of Artificial Intelligence (AI), particularly Large Language Models (LLMs), has driven a paradigm shift in software engineering. However, this progress comes at a severe environmental cost, a trend formally identified as "Red AI" [1]. The compute required for AI training has been doubling every six months since 2012, driven by the equation where computational cost is proportional to the cost of a single example, the dataset size, and the number of hyperparameters. While the training of massive models like GPT-3 and BLOOM produces hundreds to thousands of tons of CO_2eq [2], [3], the operational carbon footprint of daily, large-scale inference tasks is equally concerning.

To mitigate this, the industry is transitioning toward "Green AI," prioritizing energy efficiency and carbon reduction alongside model accuracy. A proven strategy to reduce operational carbon emissions is *temporal shifting* - executing flexible, delay-tolerant workloads during hours when the local power grid's marginal carbon intensity is lowest [4]. While tech giants have successfully implemented carbon-aware computing at the datacenter level, these interventions require deep infrastructure control.

Today, the vast majority of developers integrate AI via third-party cloud APIs (e.g., OpenAI). In this paradigm, physical infrastructure is abstracted away, turning the environmental impact of LLM inference into a "black box". Furthermore, as modern AI engineering increasingly relies on autonomous agentic workflows and orchestration frameworks (e.g., LangChain, LlamaIndex), internal API calls become deeply nested and abstracted, making manual energy tracking nearly impossible. While passive estimation tools exist to expose their Scope 3 emissions (value chain emissions as defined by the GHG protocol) [5], developers lack plug-and-play solutions to actively mitigate them.

To address these challenges, we introduce **EcoTracker**: a lightweight, application-wide carbon scheduling framework for Python. EcoTracker acts as an active orchestration layer that enables developers to implement carbon-aware AI workflows with minimal architectural overhead, bridging the gap between passive carbon telemetry and active emission mitigation at the application level.

II. Problem Statement

To enable widespread sustainable software engineering, developers need tools that reduce the carbon footprint of AI inference without introducing high cognitive load. Currently, existing carbon-aware schedulers are predominantly designed for local GPU workloads (e.g., PyTorch training loops) or heavy infrastructure orchestration, rather than API-based, stateless Cloud LLM requests. There is a critical lack of lightweight, application-layer tools that address the "black-box" nature of Cloud LLMs.

Additionally, while tools exist to estimate the hardware energy consumption of LLMs, they fall short of active mitigation. Estimating the exact power draw of closed-source LLMs requires complex heuristics based on token counts and hardware assumptions. Open-source libraries like `Ecologits` [6] successfully solve this by providing scientifically rigorous, passive estimation of base energy consumption (kWh) via process-level patching. However, passive estimation alone does not actively reduce emissions.

Finally, naive implementations of temporal shifting introduce a systemic structural risk known as the "Thundering Herd" problem. If multiple decoupled systems independently identify the same hour as the period of lowest marginal carbon intensity, they will all schedule their execution simultaneously. This creates an artificial demand spike that may inadvertently trigger fossil-fuel marginal power plants, causing a rebound effect.

Proposed Solution: To address these critical gaps, we propose **EcoTracker**, an application-wide scheduling library for LLM inference. Acting as a zero-infrastructure drop-in wrapper, EcoTracker provides:

1) **Temporal Mitigation & Rebound Jitter:** Developers can delay non-urgent workloads to low-intensity time slots, utilizing stochastic jitter to prevent artificial demand spikes (the "Thundering Herd" problem).
2) **Dynamic Model Downgrading:** EcoTracker supports execution-time model downgrading on carbon-intensive grid windows by automatically mapping selected models (e.g., OpenAI's GPT-4o) to lower-cost fallback variants, preserving telemetry on requested versus effective usage.
3) **Carbon Circuit Breaker:** A per-session budget policy monitors accumulated emissions and actively terminates execution once a user-defined gCO_2eq limit is exceeded.
4) **Rich Telemetry & Reporting CLI:** A modular telemetry layer captures normalized session records via multiple output sinks. A post-execution CLI aggregates this data across runs to report total emitted carbon, avoided carbon, and real-world equivalence metrics (e.g., equivalent kilometers driven).
5) **Provider Architecture:** EcoTracker focuses on free, keyless grid APIs (e.g., UK National Grid) while allowing developers to easily switch providers.

III. Background and Related Work

The current trajectory of Artificial Intelligence is widely described as "Red AI," characterized by the pursuit of marginally higher accuracy through exponentially larger computational capacity [1], [7]. Recent industry data from Meta indicates that for long-term deployments, the inference phase can represent up to 90% of the total energy cost [8]. This reality necessitates a shift toward "Green AI," which elevates energy efficiency and carbon transparency to primary evaluation metrics.

A. Carbon-Aware Schedulers and Telemetry

Temporal shifting is the practice of delaying flexible workloads to periods of high renewable energy availability [4]. While temporal shifting has been explored in open-source ecosystems, existing solutions fall short for application-level API developers. **Infrastructure Schedulers** (e.g., *Kube-green* [9], and *CASPER* [10]) effectively schedule heavy GPU workloads but require DevOps/SysAdmin access. **Framework-Specific Decorators** for heavy orchestration frameworks like Celery have recently emerged; however, these require massive external infrastructure (e.g., Redis databases, RabbitMQ) to function.

B. Green Architectural Tactics and Structural Rebound Mitigation

The transition to green software is often hindered by a lack of actionable design patterns [11]. Research into software energy patterns identifies multiple critical strategies, such as "Batch Operations" and "Model Optimization/Simplification" as critical strategies to reduce extraneous tail energy consumption [12] [11]. While these patterns are well-documented for traditional software, their application to LLM inference is still emerging. There is a specific need for tools that allow developers to apply these tactics at the application layer without re-architecting the underlying infrastructure. EcoTracker applies this to GenAI by allowing developers to bundle delay-tolerant tasks for execution during green hours and by simplifying the process of downgrading models.

C. Developer Experience (DevEx) and Sustainability Friction

The primary limitation in adopting sustainability tools is their negative impact on Developer Experience (DevEx) [13]. According to Greiler et al. (2022), a positive DevEx is built upon Feedback Loops, Cognitive Load, and Flow State. Current methods for carbon tracking impose high cognitive load by forcing developers to manually integrate disparate grid APIs and compute token-to-energy heuristics.

A major barrier to the adoption of carbon-aware computing is the paywalling or registration requirement of real-time grid intensity data (e.g., WattTime, ElectricityMaps). This financial and administrative friction severely degrades DevEx and halts open-source adoption.

IV. Methodology & Architecture

A. Core Features

EcoTracker implements application-level carbon-aware execution for delay-tolerant LLM workloads. Its core mechanism is temporal shifting: when a task is not urgent, the framework can postpone execution to a cleaner forecast window within a user-defined delay budget. To keep the library lightweight and easy to adopt, this scheduling is implemented entirely in Python using `threading.Timer` for synchronous functions and `asyncio.sleep` for asynchronous functions, avoiding external queueing infrastructure such as Redis or Celery. As a result, EcoTracker is highly novel because it provides **Zero-Infrastructure DevEx**. This approach targets 'stateless' carbon awareness, suitable for batch processing and background tasks where the overhead of a persistent task queue (like Celery) is unjustifiable.

Nevertheless, simple temporal shifting introduces a structural sustainability threat: the *Thundering Herd* problem. If millions of devices independently identify the same hour as the greenest, they will resume execution simultaneously, creating an artificial demand spike that triggers fossil-fuel marginal power plants. To structurally mitigate this rebound effect, EcoTracker injects **stochastic jitter** into its scheduling algorithm, randomly spreading delayed executions to ensure grid stability.

Additionally, EcoTracker introduces dynamic model downgrading. If a target execution window remains carbon-intensive, the tool automatically trades model quality for lower environmental impact by routing requests to lightweight fallbacks (e.g., `gpt-4o-mini`). To prevent runaway emissions in unpredictable agentic workflows, EcoTracker also enforces strict resource bounds via a per-session carbon circuit

breaker, which aborts ongoing executions if a defined gCO_2eq threshold is crossed.

B. High-level Design

While tools like `Ecologits` [6] successfully utilize process-level monkey-patching to provide passive telemetry for deeply nested LLM calls, EcoTracker builds upon this foundation. It transforms these passive estimates into an active orchestration layer, executing client-side temporal mitigation without requiring external schedulers. To ensure robust observability alongside active mitigation, EcoTracker implements a modular telemetry layer with multiple output sinks and normalized session records. This is complemented by a post-execution reporting CLI that aggregates telemetry across runs, presenting total jobs, emitted versus avoided carbon, and intuitive real-world equivalence metrics.

For energy estimation, EcoTracker builds on the open-source EcoLogits library rather than introducing a separate heuristic model. EcoLogits instruments supported SDK calls and estimates energy consumption in kWh and Carbon Dioxide Equivalent (CCO_2eq), which EcoTracker then combines with grid carbon-intensity data to compute baseline emissions, actual emissions, and avoided emissions. This design separates concerns cleanly: EcoLogits provides passive energy telemetry, while EcoTracker adds active mitigation through scheduling and policy enforcement.

To enable global usability, the framework supports multiple grid data providers, allowing developers to fetch carbon intensity forecasts using the free UK National Grid API or the ElectricityMaps API for worldwide coverage. Crucially, because Cloud LLM inference emissions (Scope 3) occur at the physical location of the provider's datacenter rather than the user's local machine, EcoTracker allows developers to explicitly configure the target geographic region (e.g., us-east-1). This ensures the scheduler optimizes execution against the correct local power grid, accurately mitigating the environmental impact of the externalized computation.

To maximize social impact, EcoTracker utilizes a Provider Architecture. It defaults to the open UK National Grid Carbon Intensity API for keyless, frictionless live execution, while retaining the architectural flexibility to integrate other providers in the future. By acting as a "plug-and-play" decorator, the tool honors the "Frictionless Release" and "Uninterrupted Time" DevEx factors.

EcoTracker is designed to work seamlessly with ordinary Python application code and LangChain-style agentic workflows. By natively intercepting underlying model calls through supported SDKs like OpenAI and Anthropic, the framework solves a major observability challenge: modern LLM applications often contain deeply nested model invocations that make manual energy accounting and consistent carbon-aware scheduling nearly impossible.

C. Usage

The framework is exposed through the `@carbon_aware` decorator. A developer can configure the maximum delay budget, the forecast provider, optional model downgrading on carbon-intensive grid windows, and an optional per-session carbon budget. Forecast data can be obtained from the default UK Carbon Intensity provider, from Electricity Maps, or from a deterministic CSV trace used for controlled experiments. After selecting the lowest-intensity execution window within the allowed horizon, EcoTracker may add a small stochastic jitter of up to five minutes to delayed executions in order to reduce synchronization effects between independently scheduled jobs.

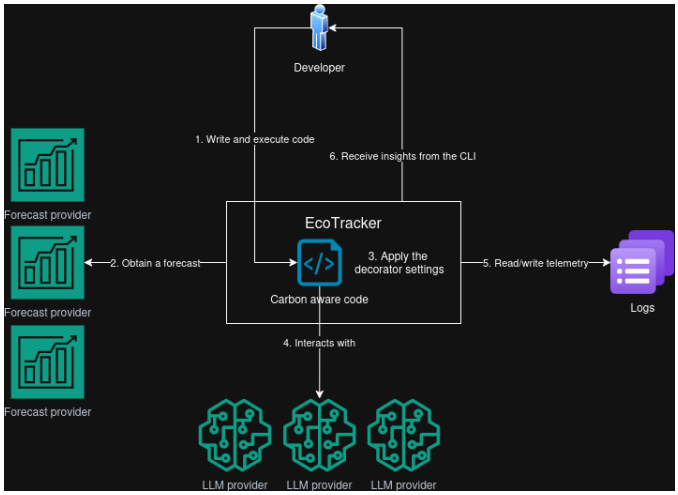

Fig. 1. High-level EcoTracker execution flow. A decorated workload is evaluated against a configured carbon-intensity forecast, scheduled within a bounded delay window, optionally modified by downgrade and budget policies, and then recorded through the telemetry and reporting pipeline.

Figure 1 summarizes the execution flow. A developer first wraps a function with `@carbon_aware` and specifies the desired scheduling and mitigation policies. EcoTracker then queries the configured forecast provider, constructs a scheduling plan, optionally applies model downgrade or carbon-budget policies at execution time, and runs the workload in the selected window. Once execution completes, the framework emits normalized telemetry records that can later be aggregated through the reporting CLI.

V. Experimental Setup

EcoTracker is evaluated in a controlled, fully reproducible offline setting. Rather than relying on live grid APIs, the main benchmark uses a static carbon-intensity trace. This design removes network variability, API availability issues, and day-to-day changes in live forecasts, ensuring that every run observes the same carbon-intensity profile. As a result, any differences in output are attributable to implementation changes rather than external noise.

The evaluation is designed to answer two questions. First, does EcoTracker's forecast-guided scheduler reduce carbon emissions relative to immediate execution under a fixed delay budget? Second, how closely does the practical scheduler approximate an oracle that has perfect knowledge of the realized carbon-intensity curve? The deterministic trace-based setup is well suited to both questions because it allows identical replay

across runs while preserving the same scheduling logic used by the library.

a) Data source: The benchmark trace contains timestamped forecast and actual carbon-intensity values in gCO_2eq/kWh for consecutive 30-minute UTC intervals. This data was obtained from the UK Carbon Intensity API by selecting the data from 2026 February 14th until March 15th. During benchmarking, each submission time is evaluated against the forecasted values available within the permitted delay horizon, while realized emissions are computed from the corresponding actual values at the eventual execution time. This separation is important: EcoTracker schedules using forecast information, whereas the oracle baseline is computed using perfect hindsight from the actual trace.

b) Benchmark workflow: For each eligible submission point in the trace, we compare three execution strategies:

- **Baseline**: immediate execution at the submission time.
- **EcoTracker**: execution shifted to the lowest forecasted-intensity window within the allowed delay budget.
- **Oracle**: execution shifted to the lowest actual-intensity window within the same delay budget.

Each scenario models a workload of 50 LLM calls with a combined energy demand of `0.0005 kWh`. For every strategy, emitted carbon is computed by combining the workload energy with the actual grid intensity at execution time. The EcoTracker strategy therefore reflects realistic forecast-guided scheduling, while the oracle provides an upper bound on achievable savings under perfect information.

c) Evaluation parameters: The main multi-day benchmark uses a maximum delay budget of four hours. The benchmark script iterates over all valid 30-minute submission slots in the trace for which a full look-ahead horizon is available, yielding 29 complete UTC days and 1,392 scheduling scenarios in total. Aggregate and daily metrics are then computed from these scenario-level results, including total emitted carbon, percentage reduction relative to baseline, delay statistics, and the fraction of oracle savings captured by EcoTracker.

d) Additional mechanism validation: Some EcoTracker features are validated through separate deterministic scripts rather than through the main multi-day benchmark. In particular, end-to-end integration, telemetry capture, model-usage tracking, and mocked OpenAI execution are exercised with the CSV-backed integration benchmark, while runtime overhead is measured independently through repeated microbenchmark runs of decorated and undecorated functions. Dynamic model downgrading and carbon-budget enforcement are also demonstrated through dedicated deterministic demo scripts. This separation keeps the primary benchmark focused on scheduling quality while still validating the broader framework behavior.

e) Execution environment and reproducibility: All experiments can be executed in a standard Python environment using the repository dependencies. No GPUs, cloud infrastructure, paid API access, or external credentials are required for the benchmark pipeline. Reproducing the results consists of installing the project dependencies and running the benchmark and analysis scripts against the repository's fixed input trace. Because the benchmark input is static, the generated CSV summaries, aggregate statistics, and paper figures are reproducible up to ordinary floating-point and platform-level formatting differences.

f) Scope and limitations: The benchmark is designed to maximize internal validity and reproducibility by using a fixed offline carbon-intensity trace rather than live provider data. Consequently, it does not capture forecast-request latency, API failures, regional differences in data quality, or other deployment-specific effects that may arise in real-world use. The results should therefore be interpreted as evidence about EcoTracker's scheduling behavior under controlled conditions, rather than as a complete evaluation of live operational performance.

VI. Results

The multi-day benchmark covered 29 complete UTC days from the fixed trace and yielded 1,392 scheduling scenarios. Each scenario modeled a workload of 50 API calls with a combined energy demand of 0.0005 kWh. For every submission point, we compared three strategies: immediate baseline execution, EcoTracker's forecast-guided scheduling policy, and an oracle with perfect knowledge of the realized carbon-intensity curve. The results show that EcoTracker consistently reduced aggregate emissions and captured a substantial fraction of the savings available under perfect hindsight.

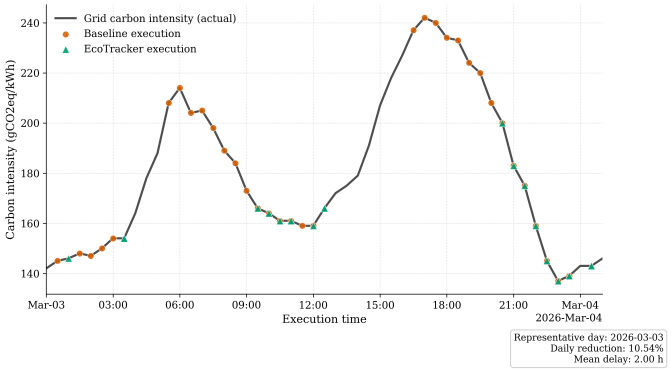

Fig. 2. Representative-day grid carbon intensity and temporal shifting. Orange circles indicate baseline execution times, while green triangles indicate EcoTracker executions shifted to lower-intensity windows.

a) Representative-day behavior: Figure 2 illustrates EcoTracker's scheduling behavior on a representative day from the benchmark trace. The gray curve shows the realized grid carbon intensity over time, the orange markers indicate when workloads would execute under the baseline policy, and the green markers show the execution times selected by EcoTracker. The visual pattern confirms the intended mechanism of the scheduler: rather than running immediately during relatively carbon-intensive periods, delayed workloads are shifted toward lower-intensity windows within the allowed delay horizon. The figure therefore provides an intuitive explanation for the aggregate savings observed across the full benchmark.

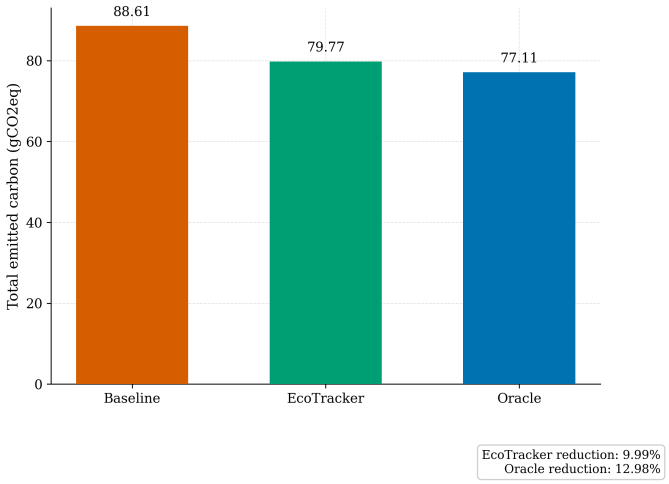

Fig. 3. Aggregate emissions across the multi-day benchmark. EcoTracker reduced total emitted carbon by 9.99% relative to immediate execution and captured 76.9% of the savings achieved by the oracle policy.

b) Aggregate emissions: Figure 3 summarizes total emitted carbon across all 1,392 scenarios. Baseline execution produced 88.61 gCO_2eq, whereas EcoTracker reduced this to 79.77 gCO_2eq, corresponding to a 9.99% reduction. The oracle policy achieved 77.11 gCO_2eq, establishing an upper bound on the achievable savings for this trace under the same delay budget. Relative to that upper bound, EcoTracker captured 76.9% of the oracle's total carbon savings. This result indicates that a forecast-guided scheduler, despite operating without perfect hindsight, recovers most of the available benefit in this setting.

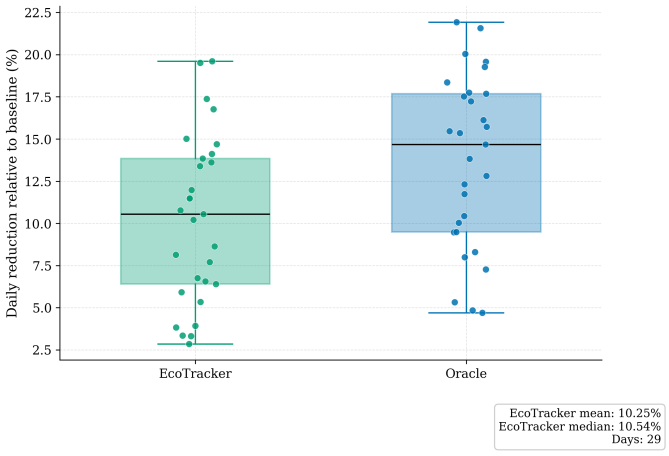

Fig. 4. Distribution of daily emissions reductions across 29 benchmark days. EcoTracker improved all 29 days relative to baseline, with mean and median daily reductions above 10%.

c) Day-level consistency: Figure 4 shows the distribution of daily percentage reductions for both EcoTracker and the oracle policy. EcoTracker achieved a mean daily reduction of 10.25% and a median daily reduction of 10.54%. Importantly, all 29 benchmark days improved relative to baseline at the aggregate day level, with no daily regressions observed. This consistency matters because it shows that the reported savings are not driven by a small number of unusually favorable days; instead, the benefit persists across the full trace.

d) Delay characteristics: The observed carbon savings were achieved under moderate scheduling delays. The median scenario delay was 2.0 hours, and the mean delay was approximately 2.05 hours under a fixed four-hour delay budget. These values indicate that the reduction in emissions does not depend on extreme postponement. Rather, the scheduler is typically able to find meaningfully cleaner execution windows within a practically bounded waiting period.

e) Interpretation: Taken together, the results show that EcoTracker's forecast-guided temporal shifting policy is effective under the controlled benchmark trace. It delivers a measurable reduction in total emitted carbon, performs consistently at the day level, and approaches the theoretical optimum defined by the oracle. From a software-engineering perspective, these gains are achieved without introducing external scheduling infrastructure, which supports the claim that application-level carbon-aware execution can be both lightweight and practically useful.

VII. Discussion

The results show that application-level carbon-aware scheduling can produce meaningful emissions reductions without requiring infrastructure-heavy orchestration. In the controlled benchmark, EcoTracker reduced aggregate emissions by 9.99% relative to immediate execution and captured 76.9% of the savings achieved by an oracle policy with perfect hindsight. At the same time, the day-level results indicate that these gains are not driven by a handful of anomalous cases: all 29 benchmark days improved relative to baseline. Taken together, these findings suggest that forecast-guided temporal shifting is a viable mechanism for reducing the operational carbon footprint of delay-tolerant LLM workloads.

A key contribution of EcoTracker is that it extends passive carbon accounting into active mitigation. Existing telemetry tools can estimate the energy and emissions associated with LLM use, but they do not by themselves change when workloads execute or how requests are routed. EcoTracker adds this policy layer directly at the application boundary. From a software-engineering perspective, this is significant because it lowers the barrier to experimenting with carbon-aware behavior: developers can introduce scheduling, model fallback policies, and budget limits without redesigning their deployment architecture around external queues or schedulers.

The results also illustrate an important practical trade-off. EcoTracker performs well relative to the oracle, but it does not eliminate the gap to the theoretical optimum. This difference is expected: the framework makes decisions based on forecast information, whereas the oracle uses realized carbon-intensity values. In other words, the residual gap reflects the unavoidable cost of acting under uncertainty. This is an encouraging result, because it suggests that most of the available benefit on the benchmark trace is recoverable without perfect foresight.

Overall, the findings support the claim that carbon-aware execution can be incorporated at the application level with relatively low engineering overhead. EcoTracker should not be viewed as a replacement for infrastructure-level carbon optimization, but rather as a complementary mechanism for developers who do not control the underlying cloud platform. Its main value lies in showing that even under the current constraints, measurable emission reductions are achievable through lightweight software intervention.

VIII. Limitations

Several limitations, however, qualify the interpretation of these findings. First, the benchmark is based on a fixed offline trace and therefore emphasizes internal validity over deployment realism. It demonstrates that the scheduling policy works under controlled conditions, but it does not measure forecast-request latency, provider outages, rate limits, or regional differences in live data quality. A live deployment may therefore exhibit weaker or less stable gains than those observed here, especially when forecast accuracy degrades.

Second, EcoTracker's lightweight scheduling model introduces operational constraints. The current implementation delays execution in-process using `threading.Timer` and `asyncio.sleep`. This keeps the framework simple and easy to adopt, but it also means that scheduled work is not durable: if the process terminates before execution, the delayed task is lost. This trade-off is acceptable for experimentation, local tooling, and some non-critical batch workloads, but it may be insufficient for production settings that require persistence, recovery, and stronger execution guarantees.

Third, the environmental benefit of temporal shifting should be interpreted together with client-side overhead. Delaying a small request for several hours may keep a client machine or server process alive while waiting for a cleaner window. For sufficiently small workloads, the idle energy consumed during that waiting period could offset part of the datacenter-side emissions savings. The present study does not model this break-even point explicitly, so the reported reductions should be understood as savings at the workload-execution layer rather than as a full end-to-end accounting of all system energy costs.

Finally, the broader applicability of the results depends on forecast availability and regional coverage. EcoTracker defaults to the UK Carbon Intensity provider because it is openly accessible and easy to use, which supports the framework's low-friction design goals. However, this convenience comes with geographic limitations: users outside supported regions may need alternative providers, and forecast quality may vary substantially across locations. Extending the framework's evaluation across multiple live providers and regions is therefore an important next step for assessing its generalizability.

IX. Conclusion & Future Work

This paper presented EcoTracker, a lightweight Python framework for application-level carbon-aware execution of LLM workloads. By combining forecast-guided temporal shifting with telemetry, model fallback policies, and carbon-budget enforcement, EcoTracker extends passive emissions monitoring into active mitigation. In a controlled multi-day benchmark, the framework reduced aggregate emissions by 9.99% relative to immediate execution and captured 76.9% of the savings achieved by an oracle policy with perfect hindsight. These results show that meaningful reductions in operational carbon emissions are achievable even when developers do not control the underlying cloud infrastructure.

More broadly, the study demonstrates that carbon-aware behavior can be introduced with comparatively low engineering overhead. Rather than requiring external schedulers or platform-level intervention, EcoTracker operates directly at the application layer through a decorator-based interface. This makes it a practical complement to infrastructure-level sustainability efforts, particularly for developers building on top of third-party LLM APIs.

Several directions for future work follow naturally from the present study. First, a fuller end-to-end efficiency analysis is needed to determine when delayed execution remains beneficial after accounting for client-side idle power consumption. Second, the current in-process scheduling model could be extended with optional lightweight persistence mechanisms to improve durability for longer-running or production-facing workloads. Third, broader live-provider evaluation across regions would help assess how well the framework generalizes under real-world forecast uncertainty and differences in data availability. Finally, the policy layer itself could be expanded through richer scheduling heuristics, more systematic evaluation of model downgrading behavior, and tighter integration with additional LLM application frameworks.

Overall, EcoTracker provides a concrete step toward more sustainable AI-enabled software. Its main contribution is not only that it measures the carbon implications of LLM usage, but that it shows how software can respond to them directly through lightweight application-level orchestration.

References

[1] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green ai," *Communications of the ACM*, vol. 63, pp. 54–63, 11 2020.

[2] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, "Carbon emissions and large neural network training," *arXiv preprint arXiv:2104.10350*, 2021.

[3] A. S. Luccioni, S. Viguier, and A.-L. Ligozat, "Estimating the carbon footprint of BLOOM, a 176b parameter language model," *Journal of Machine Learning Research*, vol. 24, no. 253, pp. 1–38, 2023.

[4] A. Radovanović, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care *et al.*, "Carbon-aware computing for datacenters," *IEEE Transactions on Power Systems*, vol. 38, no. 2, pp. 1270–1280, 2022.

[5] Greenhouse Gas Protocol, "Corporate value chain (scope 3) accounting and reporting standard," *Chapter*, vol. 2, p. 11, 2011.

[6] S. Rincé and A. Banse, "Ecologits: Evaluating the environmental impacts of generative ai," *Journal of Open Source Software*, vol. 10, no. 111, p. 7471, 2025. [Online]. Available: https://doi.org/10.21105/joss.07471

[7] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019, pp. 3645–3650.

[8] C.-J. Wu, R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. A. Behram, J. Huang, C. Bai *et al.*, "Sustainable AI: Environmental implications, challenges and opportunities," in *Proceedings of Machine Learning and Systems (MLSys)*, vol. 4, 2022, pp. 795–813.

[9] D. Bianchi and Contributors, "kube-green: A Kubernetes operator to reduce CO2 footprint of your clusters," https://github.com/kube-green/kube-green, 2022, accessed: 2024-05-20.

[10] A. Souza, S. Jasoria, B. Chakrabarty, A. Bridgwater, A. Lundberg, F. Skogh, A. Ali-Eldin, D. Irwin, and P. Shenoy, "Casper: Carbon-aware scheduling and provisioning for distributed web services," in *Proceedings of the 14th International Green and Sustainable Computing Conference*, 2023, pp. 67–73.

[11] H. Järvenpää, P. Lago, J. Bogner, G. Lewis, H. Muccini, and I. Ozkaya, "A synthesis of green architectural tactics for ML-enabled systems," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, 2024, pp. 130–141.

[12] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, vol. 24, pp. 2209–2235, 2019.

[13] M. Greiler, M.-A. Storey, and A. Noda, "An actionable framework for understanding and improving developer experience," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1411–1425, 2022.