Dynamic Tagging: Tracking Energy Usage on LLMs

CS4991 Capstone Report, 2024

Tu-Yen Dang
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
frk3nx@virginia.edu

ABSTRACT
With the recent uprise in AI and high-level computing processes, there is consequently an increase in energy usage. To increase user awareness and help determine sustainable yet efficient energy options, we have developed a program that dynamically tracks the energy used in each instance of a given program. This was done through simulating a user's code on a given computing environment, processing its workflow, and calculating the energy based on the cost of the instance, effort needed on a task, and the time it took to complete the task. We chose Python as the primary programming language for this method since many machine learning (ML) models are available and manipulable with this language. The dynamic tagging was successfully applied to one of our larger projects and showed an accurate monitored log for energy consumption of the process. Though we produced a successful system, there are still many opportunities for the system to be more usable and accurate, like creating a more readable logging system for users and a smarter optimization process.

1. INTRODUCTION

Recently, there has been an uprise in finding and using faster, stronger, and more efficient computational power, especially with the rise of AI. In fact, the AI market size is projected to reach $407 billion by 2027, which is astronomical compared to the estimated value of it in 2022 ($86.9 billion) [1]. As these trends progress in the software industry, there is consequently a higher environmental footprint tied to it. In fact, it was found that training a large-scale AI model could produce over 600,000 pounds of carbon dioxide, nearly five times the lifetime emissions of an average US manufactured car [2]. From intensive computations, mass database usage, to complex analytical processes, it is crucial to keep track of the impact software programs have on the environment and make more sustainable decisions, including computing power, electricity, and carbon emissions.

However, this can be difficult since not many resources are available that can compute energy usage and emission rates. In order to solve the problem at hand, a Dynamic Tagging system was designed with the intention of tracking energy usage and emissions of code so that programmers can monitor the usage rates of their workflows on each instance. This system will tag and track each task on a program, and return its estimated energy usage, and will suggest optimization solutions for companies to make more environmentally-conscious choices on running programs.

2. RELATED WORKS

Despite software energy tracking being a fairly recent yet important factor in programming, tracking energy usage is not new. Before software, there was still an economic need to track tools that relied on

electricity like lighting. Tracking electricity use would be essential for utility companies to charge homes appropriately for how much they used. Thus, Thomas Edison created an electric meter in 1881, which measured the amount of electricity used by calculating the difference in weight of a copper strip before and after a billing period; this strip had the current pass through an electrolytic cell, which would then cause a deposition in the copper [3]. This is similar to the current dynamic tagging system since it measures the energy usage. However it is also different since it focuses on the hardware, while the dynamic tagging system is purely for software purposes in the modern day, where in-person tracking may not be accessible.

One modern-day related work is a Python package called codecarbon. Codecarbon estimates the carbon emissions a program uses when implementing its package into said program. It measures both local and cloud -based programs and can even split the program into different tasks so users can see which specific part of their code is most environmentally friendly [4]. However, the package relies on estimations that may not be fully accurate, and only records carbon emissions. This disregards the main point of tracking energy usage along with the emissions to offer optimal solutions for users.

A tool found to handle some of codecarbon's insufficiencies was AWS's Customer Carbon Footprint Tool, which measures the emissions, provides a user friendly in-depth review of emission rates for programs, and even provides forecasting and future planning services [5]. Though this product seemed to have almost every key point desired, we still took a different direction with another design. AWS's tool may be powerful, but it generalizes the emission reviews over a monthly span, whereas we wanted to focus on a program that would track the program dynamically. Having it do so would allow for recurring instances such as people querying an AI model to be traceable on its own.

3. SYSTEM DESIGN

The Dynamic Tagging tracker system was intended to be used with Python programs and was specifically created for the User Hospitality Services (UHS) team at Leidos, who were building a large language model (LLM) to process user lookups and find credible and desired databases. Though it is projected to integrate with UHS, it was also proposed to function alongside many other projects.

3.1 Modeling a Workflow

To conform with each project's needs, this system follows a simulated version of a given program. In this program, a function is represented by a Task object, which contains how much effort is needed for the task to be completed, and the time it took to complete (rate). What the task is being run on is represented by the Compute Instance object, which has a Resource property that stores the costs of using that specific instance and the capacity of how much it can perform.

The Compute Environment represents a group of instances that can all be used to perform tasks. This environment can perform tasks in many two ways: monitoring and optimizing. Monitoring will simply run the simulation assuming each task is ordered; while optimizing works around the premise of determining which compute instance would be best to use first before moving on to more energy-demanding ones. Finally, two objects are responsible for processing the workflow, and logging it: the Workflow and Data Product class. A Workflow is able to process a list of tasks that is associated with it in an environment, and each Data Product is able to follow one or more tasks and record its history of performance and can then record it

in a log file when desired. Figure 1 and Figure 2 go in depth with each component of the system and how they interact with one another.

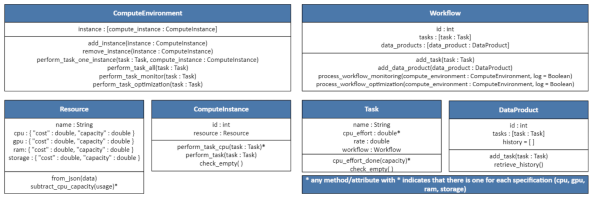

Figure 1. Description of each class in dynamic tagger

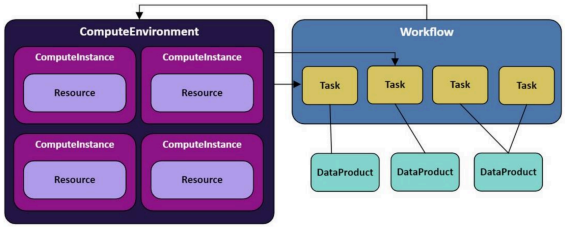

Figure 2. Representation of Class Interaction

3.2 Calculating Performed Energy Rates

To calculate energy usage, four computer specifications are being monitored in the simulation: CPU, GPU, RAM, and Storage. The Task object describes how much effort each specification requires in order to fully execute, and each Compute Instance object has resource properties that list each specification's capacity (maximum amount of usage), and cost (cost of running the task on said instance). The program intends to measure energy values of these specifications in rates per unit effort, and the energy "effort" of some program being performed is represented by the combination of the rate of time the resources were needed for, and how hard the work being done will be.

To calculate energy usage of a task, we used the following equation to represent a task being performed: *Rate of task being done * instance's capacity used * cost of using instance's resource.* While the task is being performed, the compute instance's capacity values will update with it, and the task effort required to run will, as well.

3.4 Running the Program

With its current implementation, the tagging system requires some user pre-emptive information to work. Initially, only one version was to be created: running with a decorator. However, when testing its compatibility with another project, we ran into an issue regarding the structure. So, a second method was formed to run the Dynamic Tagging program. The first method is with a decorator on functions. This method would require the user to attach the tag_tracker() decorator into the functions (or Tasks) of their programs, and an external JSON file with a list of compute instances would be needed. For the CCH team, one was provided. In the future, the decorating function should additionally read in values to build a task while the function is running. At the moment, the decorator is intended to be used to identify the function's name and correlate it to a list with its values to create a representative Task object. To process the workflow, users should then run the execute tracking() function at the very end of their code. Figure 3 depicts how using the decorator would work for further explanation.

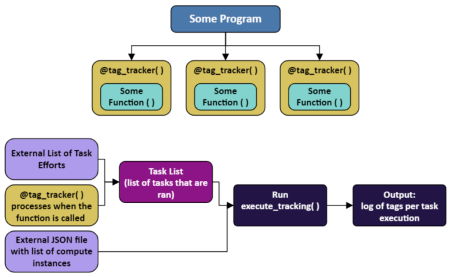

Figure 3. Running Dynamic Tagging with a Decorator

The second method to use the program is fully separate from the user's program, and only requires given files. Now what is needed from the users is an ordered list of what tasks will run, and the compute instance JSON file.

To process the workflow, users should then run the execute_tracking() function; nothing else is needed to run. Figure 4 provides further clarification on the steps.

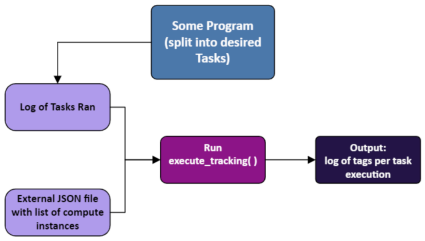

Figure 4. Running Dynamic Tagging with Pre-Emptive Information

4. RESULTS

The dynamic tagging tracker system was used for internal Leidos projects. It was seen to be completely and seamlessly compatible with the CCH team's vza-cold-product workflow, a project related to tracking night lights data depending on the day. Originally, this system also had the intention to work with the UHS team's LLM database lookup process, as well. However, because the LLM was not completed or testable at the time, it cannot be fully seen as successful yet. Both projects revolve around having an efficient but minimally power-consuming workflow. So, this tagging system was used in conjunction to record each instance the program would be run and record the values needed.

At the moment, the log specifies which instance is performing a task and the usage values for each compute specification, allowing for users to monitor each value, if needed. However, if there were a use case that required a different log format, the program is incredibly flexible with output.

When using the monitoring versus optimization options, monitoring will go through the instance list in its given order, while optimization will sort the list in a cost-order before running the tasks. In return, optimization will prioritize initial tasks with the least costing instances, leaving the more energy-costing instances for the end. Though other methods may yield other outcomes, this one is proven to optimize workflows where tasks do not require all instances to work.

5. CONCLUSION

The Dynamic Tagging system offers a significant advancement in the monitorization and optimization of energy usage for software processes, particularly in the context of machine learning and large language models. By providing real-time tracking of hardware consumption, the system allows users to make more informed and sustainable decisions regarding their computational workflows. Its flexibility with different simulation modes allows it to be used with various project needs. Ultimately, this project is an important step toward increasing awareness in computational energy consumption, offering a tool for those looking to minimize their environmental impact while maintaining high levels of computational performance.

6. FUTURE WORK

Despite having a fully functional application, there were many next steps that could further the potential for this program. One would be to work with Codecarbon, mentioned in related works. This application was missing components we were looking for, but if incorporated into the current tagging system, energy efficiency and emissions rates could be further tested against one another. Another improvement would be to output something more readable for users. The current application simply outputs a log of every instance performed and lacks summary skills to make it easier for users to understand what they are reading. Having a system after the logging to review and summarize findings may be helpful for others interested in this product. Finally, one other possible expansion would be to use some level of ML to determine optimal energy rates. A very

simple decision-making algorithm but using ML to determine optimal choices can be more accurate and reliable.

7. ACKNOWLEDGMENTS

I would like to thank my manager and team member Ian Paynter and Peter Boucher for their guidance, support, and contributions throughout this project. I would also like to thank Leidos and the User Hospitality Team for giving me the opportunity to work on this project and for providing the resources and environment to complete this work.

REFERENCES

[1] Katherine Haan. Forbes. 2024. 24 Top AI Statistics and Trends In 2024. (June 2024). Retrieved September 25, 2024 from https://www.forbes.com/advisor/business/ai-statistics/

[2] Karen Hao. MIT Technology Review. 2019. Training a Single AI Model can Emit as Much Carbon as Five Cars in Their Lifetimes. (June 2019). Retrieved September 25, 2024 from https://www.technologyreview.com/2019/06/06/239031/training-a-single-ai-model-can-emit-as-much-carbon-as-five-cars-in-their-lifetimes/

[3] Smart Energy International. 2006. The History of the Electricity Meter. (June 2006). Retrieved September 25, 2024 from https://www.smart-energy.com/features-analysis/the-history-of-the-electricity-meter/#:~:text=Thomas Alva Edison

[4] CodeCarbon. 2021. CodeCarbon: Track and Reduce CO2 Emissions from your Computing. Retrieved September 25, 2024 from https://codecarbon.io/

[5] Amazon Web Services. 2024. Understanding the Customer Carbon Footprint Tool. Retrieved September 25, 2024 from https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/ccft-overview.html