Green Shift Left - Evaluating energy efficiency on the web with static analysis

Jort van Driel
TU Delft
jortdriel@tudelft.nl

Dorian Erhan
TU Delft
derhan@tudelft.nl

Weicheng Hu
TU Delft
weichenghu@tudelft.nl

Giannos Rekkas
TU Delft
grekkas@tudelft.nl

***Abstract*—Despite a growing awareness of sustainable software practices, the web remains largely unoptimised and understudied from an environmental perspective. The modern web is now responsible for approximately 3.7% of global greenhouse gas emissions. Given the ubiquity of JavaScript across the web ecosystem, even small inefficiencies can add up to significant environmental costs. Existing development tools lack mechanisms to provide real-time feedback on the energy impact of code. To address this gap, this research presents an empirical analysis of JavaScript coding patterns and proposes a low-overhead, open-source static analysis solution that integrates with CI/CD pipelines. We tested improvents for a number of anti-patterns and built an ESLint-based static analysis tool to flag this inefficient code in real time. In our experiments, we found that most of the proposed improvements did not result in significant decrease of the energy consumption. However, caching expensive identifiers such as `querySelectorAll` leads to dramatically lower energy consumption. On the other hand, surprisingly, using webp instead of gifs and css animations instead of canvas actually lead to higher or similar energy consumption. By leveraging established energy-aware design patterns and providing actionable insights, this work aims to empower developers to write more efficient, sustainable code and push for the integration of energy-aware tools into existing continuous integration pipelines.**

***Index Terms*—JavaScript energy efficiency, Web optimization, Static analysis, ESLint, Sustainable software development, Coding patterns, Web performance**

I. INTRODUCTION

Every year, hardware advances bring faster processors and larger memory capacities, but no matter how much the hardware improves, developers always tend to add features to push their software to the limits of what is possible on the given hardware. This has led to increasingly bloated software, where efficiency and optimization often take a back seat to rapid development and feature richness [1]. This shift has resulted in an industry-wide lack of concern for efficiency and simplicity, with an ever-increasing demand for performance optimization despite the use of faster CPUs and larger memory systems. In stark contrast, the Apollo guidance computer successfully sent men to the moon with just 4KB of RAM, while today a single browser tab can consume gigabytes of memory with little scrutiny.

Although the Web is a fundamental part of modern software development, research on its energy efficiency patterns remains scarce [2]. Web applications increasingly rely on excessive JavaScript frameworks, heavy multimedia content, and inefficient rendering techniques, all of which require significant processing power. As a result, the Internet's energy consumption has skyrocketed and now accounts for approximately 3.7% of global greenhouse gas emissions, equivalent to all of the world's air traffic [3]. Given the scale of its environmental impact, optimizing the Web's energy efficiency should be a top priority.

At the heart of this challenge is JavaScript, which has become the cornerstone of modern software development. From front-end interfaces to back-end services to mobile apps, the language is ubiquitous across the web development stack. According to Stack Overflow's 2024 Developer Survey [4], JavaScript is the most widely used programming language for the tenth year in a row. Its ubiquity means that even small inefficiencies in JS code can add up to significant energy costs across millions of devices and users. JavaScript is a multi-paradigm language that supports object-oriented, imperative, and declarative programming styles. Besides the fact that it is an interpreted language, another contributor to performance issues in JavaScript is its rich and often functionally redundant APIs. Multiple coding constructs can provide the same functionality, but differ significantly in performance [5]. Developers often use sub-optimal idioms, inadvertently introducing energy inefficiencies.

Inefficient software directly contributes to higher energy consumption. While software does not independently emit CO2, it does drive hardware resource usage [6]. An analysis of the COP28 climate conference website found that bloated web code, such as unused scripts, oversized images, and heavy third-party libraries, caused excessive data transfer and processing [7]. Despite growing awareness of software sustainability, developers still lack feedback on the energy impact of their code. Existing tools, such as linters and static analyzers, focus on style, correctness, and maintainability, with little to no attention paid to energy efficiency. As a result, energy-related anti-patterns go undetected, and developers may be unaware that certain idioms or practices could have a measurable environmental impact.

This creates a gap in the focus on energy awareness in these everyday developer tools, such as linters. Many developers may use static code analysis to determine which parts of their code can be optimized for time complexity and concurrency. Although time complexity can indeed serve as a predictor of energy consumption in some cases [8]. However, there is no existing literature to suggest that this correlation is universal. Research suggests that energy consumption is influenced by factors beyond algorithmic complexity, such as hardware architecture, compiler optimizations, and system-level interactions [9], [10]. For example, the energy efficiency of an algorithm can vary depending on how it uses the memory hierarchy and manages data movement [11], so we cannot conclude that time complexity is proportional to energy consumption.

Building upon the existing work, our research aims to address these gaps by developing a low-overhead, open-source static analysis solution that integrates into existing CI/CD pipelines. This tool will provide developers with immediate, actionable insights into energy-inefficient code segments, facilitating energy-aware development practices in the web ecosystem. To bridge this gap, we propose to use ESLint, a widely used static code analysis tool for JavaScript, to identify energy hotspots within codebases. By developing custom ESLint rules that focus on patterns known to impact energy consumption, we hope to provide developers with actionable insights to help them make trade-offs between energy efficiency and other factors.

Key Takeaways. The results of the experiments show that the vast majority of the selected code patterns have no significant change in energy consumption. Only one anti-pattern shows a clear and statistically significant increase in energy consumption: the repeated use of expensive identifiers that access expensive DOM elements such as `querySelectorAll`. Caching these references results in significantly lower energy consumption. Interestingly, two anti-patterns, 'Avoid using canvas' and 'Avoid using GIF', show lower energy consumption than their supposedly optimized counterparts. This highlights the importance of empirical validation in promoting energy-efficient coding practices.

All code used in this study, including the linter plugin, experiments, testing framework, and results, is open source and available to the public via a GitHub repository[†].

II. Background and Related Work

Energy patterns are reusable solutions designed to reduce the energy footprint of software applications. Initially developed for mobile applications, recent studies have investigated their applicability to web development. Rani et al. conducted an exploratory study to assess the transferability of mobile energy patterns to the web domain. They identified 20 patterns that could be adapted and validated through interviews with six expert web developers. The study revealed that developers were particularly concerned about functional anti-patterns and emphasized the need for guidelines to detect such patterns in source code. Empirical evaluations of patterns such as Dynamic Retry Delay (DRD) and Open Only When Necessary (OOWN) showed that while OOWN led to energy savings, DRD did not show a significant reduction in energy consumption [2].

[†]https://github.com/JortvD/cs4575-g5-p2

Static analysis involves examining code without executing it to identify potential issues, including bugs, security vulnerabilities, and performance bottlenecks. Linters are tools that perform static analysis to enforce coding standards and detect errors. In the JavaScript ecosystem, tools like ESLint and JSHint are widely used to maintain code quality and consistency. ESLint is highly configurable, allowing developers to define and load custom rules [12]. JSHint, a fork of JSLint, offers a flexible approach to static code analysis, enabling integration into various development workflows [13].

Recent research has explored extending static analysis techniques to estimate and monitor the energy consumption of software. For instance, Brosch investigated how the use of static code analysis influences the energy consumption of software, highlighting the potential for integrating energy consumption metrics into development workflows [14]. Bangash et al. proposed a static-analysis approach that estimates the energy consumption of API usage in smartphone applications, eliminating the need for test case execution. Their findings indicate a positive correlation between their static analysis estimates and hardware-based energy measurements [15]. Similarly, Li et al. proposed a static-analysis approach to estimate the energy consumption of API usage in smartphone apps, eliminating the need for test case execution [16]. These studies suggest that incorporating energy consumption considerations into static analysis tools can provide developers with actionable insights to optimize their code for energy efficiency.

Further research on the subject has produced a variety of similar tools. EnergyAnalyzer is a static analysis tool developed by Wegener et al. that estimates the energy consumption of embedded software by utilizing worst-case execution time (WCET) analysis techniques. It has been validated across various benchmarks, demonstrating less than 1% difference compared to empirical energy models validated on real hardware [17]. Sousa et al. presented a technique to statically estimate the worst-case energy consumption for SPLs. By combining static analysis with worst-case prediction, the approach analyzes products in a feature-sensitive manner, achieving a mean error percentage of 17.3% in energy consumption estimation [18]. Grech et al. developed methods for statically analyzing LLVM Intermediate Representation (IR) programs to estimate energy usage, bridging the gap between high-level code structures and low-level energy models [19]. Huizhan et al. explored using static WCET analysis to guide dynamic voltage scaling (DVS) for energy optimization in real-time applications, demonstrating the potential of compiler-directed strategies in reducing energy consumption [20].

Beyond academic research, various initiatives have emerged to promote practices for **sustainable development**. The Collectif Conception Numérique Responsable (CNumR) is a French collective dedicated to responsible digital design. CNumR provides tools, certifications, and training to promote eco-friendly web development practices. Their work includes the development of best practices and guidelines to reduce the environmental impact of digital services [21]. Similarly, the Green Code Initiative advocates for integrating energy efficiency considerations into software development processes, emphasizing the importance of sustainable coding practices.

Despite these advancements, challenges remain in integrating energy-efficient practices into mainstream web development. There is a notable lack of standardized guidelines and tools that seamlessly integrate into developers' workflows and provide real-time feedback on the energy implications of their coding choices. In addition, the effectiveness of existing energy patterns and tools varies, underscoring the need for continuous evaluation and refinement.

III. Methodology

In this section, we justify why we chose ESLint as a linter for building our own rules, and how we set up our experiment and result processing.

A. Language and Linter Selection

An academic study measuring 27 languages found that performance and energy efficiency are not always aligned — a "faster" language is not necessarily the most energy efficient [22]. So, the choice of programming languages is not limited. **JavaScript** was chosen as the focus for this experiment. One of the motivations is that JavaScript is the most widely used of all programming languages [4].

Therefore, as the most popular linter for JavaScript, ESLint is used more often compared to other linters. Also, unlike some other languages, like for example, Python, where we would have to manually implement an AST with Pylint, ESLint already provides this functionality in its infrastructure, which simplifies our efforts. All of this leads to a better ecosystem for extending ESLint. We also write basic unit tests for the linting rules to make sure they work as desired before running the experiments.

B. Selected Design Patterns

In our research, we found several suggested design patterns for web design, as well as some static code analysis rules that have already been implemented [23]. Here are the anti-patterns and design patterns we chose to implement analysis rules for, and our motivation for choosing them. We also explain how we set up similar code samples that either comply or violate these rules to find the difference in energy consumption between the two scenarios.

- **Use lazy attribute**: We implement a rule that warns when using a `<img>` element that is not set to load lazily. We found from the MDN web documentation that this can reduce energy consumption by delaying loading an image until it is visible [24]. Our rule-compliant experiment imports 16 images using `loading=lazy`, and our violating sample uses `loading=eager`, the latter being the default behavior.
- **Preload link**: Adding `rel=preload` to a `<link>` element for all documents you know will be loaded on your webpage was introduced as a form of speculative loading, and is said to improve performance [25]. We are interested in whether this performance increase has an impact on energy consumption, so we have two samples that import a stylesheet, one with a preload link and one without.
- **Avoid using canvas**: When loading a webpage, about half of the energy is used for rendering [26]. Developers use `<canvas>` in cases that require detailed graphical control or complex animations, but according to a previous study, CSS can have the best performance in terms of accuracy and precision among different web technologies [27]. We have set up a basic experiment comparing a canvas with a CSS animation to see what the effect is on energy consumption.
- **Avoid using GIF**: Animated GIFs were introduced in 1995, and there are more modern encodings available that are more compressed [28]. Inspired by design pattern 4002 from CNUMR, we experiment with how the energy consumption of the more modern `.webp` format compares to using `.gif`.
- **Use standard fonts**: Different font sizes have an impact on client-side resource consumption, including CPU usage, memory allocation, load time, and energy consumption [29]. For this reason, we have implemented a rule that checks for custom fonts and issues a warning for fonts that are not in standard libraries. In our experiments, we load either a standard font or a custom font.
- **Avoid logging often**: `console.log()` is not standardized, so its behavior is browser dependent. Depending on the browser engine, its functionality may be either synchronous or asynchronous. Because JavaScript is single-threaded [30], multiple logging calls may result in blocking behavior. This may result in a higher energy overhead compared to bundling multiple objects into a single logging operation.
- **Use document fragment**: When appending multiple elements to your HTML document using JavaScript, there are two options: appending directly to the document, and the lesser-known option of appending to a temporary document fragment that is then added to the document at once. The latter option can be more performant because it does not require the document to be refreshed many times [31], [32]. We create an experiment where we add $100,000$ elements either directly to the document or via the document fragment.
- **Use intersection observer**: The Intersection Observer API can be used to create a listener for a part of the web page that is scrolled into view [33]. This allows you to load only when something is in view. The alternative solution is to use a scroll event listener to check if something is in view each time the user scrolls. We will create two experiments to compare these two behaviors.
- **Use request animation frame**: The `requestAnimationFrame` functionality has been introduced to automatically update animations at the same rate as the display refresh rate. This API also introduces other changes, such as pausing when the page is in the background, which improves battery

life [34]. We are testing to see if there is a difference in energy consumption when used in the foreground between `requestAnimationFrame` and the original `setInterval` behavior.

- **Avoid expensive identifiers**: A simple optimization is to cache calls that are expensive to make, such as finding items by some identifier using `querySelectorAll`. We create a simple experiment that toggles some items 500 million times, either calling `querySelectorAll` every time or only once before.
- **Avoid resizing images**: When an image is loaded into a web page, but then needs to be displayed at a different size, the image must be resized, causing some performance overhead. This is especially true when many images are displayed [28]. CNUMR's design pattern 34 recommends that images not be resized using their `width` and `height` attributes. We create an experiment to see if resizing versus not resizing has a significant impact on energy consumption.
- **Respect the bfcache**: Finally, the back/forward cache allows for faster navigation when returning to a previously visited page by caching the resources and heap of that page. However, if an `unload` or `beforeunload` event listener is set, the bfcache is not used, negating this performance benefit [35]. We test how this affects the energy consumption by testing it on a web page with a complex `canvas` scene.

C. Experimental Setup

The experiments were conducted on an HP ZBook Studio G5 118L5ES running Ubuntu 24.04. This laptop has an Intel i9-9750H CPU, an Nvidia Quadro P2000 GPU, and 16 GB of RAM. To minimize background noise when running the experiments, we make sure that the configuration settings are kept consistent across all experiment runs. When running the experiments, we make sure that the laptop is in a controlled environment, which includes

- No unnecessary applications/services are running.
- All external hardware is disconnected from the device.
- Notifications are turned off.
- Both WiFi and Bluetooth are turned off.
- The screen brightness of the display is set to 100%.

D. Experimental Procedure

The goal is to measure the difference in energy consumption between inefficient JavaScript coding patterns and their supposedly optimized counterparts. For each comparison, we test a piece of code that follows a known anti-pattern against a more efficient version based on established coding best practices. For example, we take JavaScript code that loads GIFs and compare it to code that loads more efficient formats, such as WEBP. All measurements are performed using EnergiBridge [36], which is an open-source cross-platform energy measurement utility[†]. We chose joules as the unit of measurement because it provides an absolute amount of energy measurement that tells us exactly how much energy was used in total, independent of time variations.

[†]https://github.com/tdurieux/EnergiBridge

For this study, we use Chromium because it is the basis for many of the most widely used browsers, including Google Chrome, Microsoft Edge, and Opera, making it a good representative choice [37]. Testing on Chromium ensures that the results are relevant to a wide range of browsers built on the same engine. We use Chromium version 137.0.7104.0.

For each design pattern described in III-B, we perform an experimental test by launching Chromium and separately measuring the energy consumption on an HTML page containing the anti-pattern and another HTML page containing the optimized version. To reduce bias, the order of the web pages is randomized during each experiment. Because hardware temperature can affect energy consumption, the very first run (used as a warm-up) is excluded from the final analysis. In total, we repeated this step 30 times for each rule.

Within each step, the following actions are performed:

1) Launch Chromium through EnergiBridge, getting 5 measurements per second.
2) Open a new tab containing the test page (either the anti-pattern or the optimized variant).
3) Wait 5 seconds to make sure all resources are fully loaded.
4) Close the tab.
5) Wait 1 second for the tab to close properly.
6) Repeat steps 1-5 for the alternate variant (the one not tested in the previous step).
7) Close Chromium and wait 5 seconds before proceeding to the next step.

After each run, we reset all generated caches and user data before proceeding to the next run.

One of the rules involves testing the efficiency of code that is compatible and incompatible with the back-forward cache (bfcache). The bfcache is a browser optimization that stores pages in memory so that they do not have to be reloaded, allowing for instant backward and forward navigation [35]. Because of this, the methodology for this particular rule has an extra step where we navigate away and then back after loading the web page to make sure we are measuring this functionality correctly.

E. Output postprocessing

After running the experiments, we perform a number of data processing steps to obtain our results. First, we select only the samples from 1 second before the web page is opened in the browser, to ensure that we do not miss any relevant data because the web page may not be opened at exactly the millisecond we expect. The selection is made until the web page is closed, which is 11 seconds for all experiments except the bfcache experiment, where it is 17 seconds. We then sum the positive increases in PACKAGE_ENERGY (J) over this time period, which gives us an estimate of the CPU and measurable motherboard energy consumption per experiment. To remove any outliers, we use the Inter-Quartile

Range(IQR) method to remove samples outside the upper and lower $1.5 * IQR$ whiskers. We manually check that this does not remove too many samples, but in our case, we did not accept any measurements that were rejected by the IQR calculation.

IV. Results

In the table I, we show the results of the experiment. **All measurements are made in joules.** To avoid confusion, a violating rule corresponds to the anti-patterns described in III-B, while a compliant rule represents its corresponding optimized version. Our static analysis tool checks for the presence of these violating rules, that is, it analyzes the code for energy-inefficient anti-patterns, raises a warning if any are found, and suggests using the optimized alternative.

We compute the Shapiro-Wilk test per experiment to see if the distribution of samples follows the normal distribution. We use the typical threshold of 0.05; any values above the threshold imply that the experiment is normally distributed. If this is the case, we apply Welch's two-sided t-test to see if the change in the mean between the two distributions is significant, otherwise, there may still be a significant change in mean that we can find with the two-sided Mann-Whitney U test.

Following the result, we provide an analysis of the percentage changes in joules before and after applying the rules in Table II. Here, in addition to the mean difference and the percentage mean change, we also apply Cohen's d to see if we can classify this change as large by including the variance of both distributions in the difference calculation.

Finally, we visualize the tables I and II using bar charts to make the comparison easier. The blue bar in Figure 1 represents the energy consumed when rules are violated, and the orange bar represents the energy consumed when rules are not violated. Figure 2 shows the percentage change in energy consumption after the rules are followed.

From Table I we can see that, except for 'Use request animation frame', all other experiments follow a normal distribution, which means that t-test p-values can be used in most cases. 'Use request animation frame' is not normally distributed, but it has a U-test p-value of < 0.001, and its shape is close to a normally distributed graph according to its violin plot VII-A, so we see it as a weaker statement of statistical significance, suggesting that energy consumption is lower for the compliant rule.

Here, the rules that are statistically significant (p-value $<$ 0.05) are summarized:

- **When energy consumption is higher if we comply with the rule (the "anti-pattern" is more efficient):**
  - **Avoid using canvas**
  - **Avoid using GIF**
- **When energy consumption is higher if we violate the rule (the optimized pattern is more efficient):**
  - **Avoid expensive identifiers**

Finally, we conduct a Cohen's d effect size analysis, the data can be found in Table II. An absolute effect size of < 0.8 is considered small; anything bigger than 0.8 is considered a large effect size [38]. Both results from the t-test and the effect size analysis leave us with 3 rules that have a larger significance:

- **Large Effect Sizes (absolute Cohen's d $\geq$ 0.8):**
  - **Lower Energy Consumption**
    - ∗ Avoid expensive identifiers (Cohen's d = 25.115)
  - **Higher Energy Consumption**
    - ∗ Avoid using canvas (Cohen's d = -1.019)
    - ∗ Avoid using GIF (Cohen's d = -3.526)

From all the results, we can conclude that avoiding expensive identifiers leads to dramatically lower energy consumption. On the other hand, avoiding gifs and canvas leads to a higher energy consumption. The remaining rules do not seem to make a significant difference.

V. Discussion and Limitations

Results discussion. The results of our experiments present some unexpected findings. The general assumption was that known anti-patterns would result in higher energy consumption compared to their optimized counterparts. However, most of the cases did not show any statistically significant difference in energy usage. Furthermore, there were more cases where anti-patterns were more energy efficient than their counterparts than the inverse.

The only rule that demonstrated a clear improvement in energy efficiency when optimized was "Avoid expensive identifiers". Expensive identifiers are functions that repeatedly access the DOM, such as `querySelectorAll`. Since DOM access is a relatively costly operation in terms of both performance and energy, this finding aligns well with what we might expect. Reducing redundant DOM queries minimizes computation, leading to improved energy efficiency.

Interestingly, a similar pattern, 'use document fragment', did not display the same behavior. This idiom aims to reduce the number of direct DOM appends by, for example, batching multiple `appendChild()` calls together using a document fragment [31]. However, the energy consumption remained nearly identical between the anti-pattern and the optimized version in this case. One possible reason for this could be that repeated DOM querying involves complex traversal of the DOM, which is a complex and expensive computation within the browser. In contrast, appending elements, even multiple times, might not trigger as much layout recalculation or reflow until rendering occurs. Another possible justification, which applies to all other patterns as well, is that modern JavaScript engines may already optimize certain known anti-patterns under the hood. This could diminish the practical impact of some coding optimizations.

Another rule where the optimized pattern did not have better results was 'respect bfcache'. In this experiment, we included an expensive canvas animation in the HTML to simulate a heavy rendering process that could have benefited from the back-forward cache. However, this setup does not reflect typical real-world cases, where pages have to make multiple

TABLE I
STATISTICAL RESULTS OF ENERGY CONSUMPTION (J) FOR COMPLAINT AND VIOLATING SAMPLES PER RULE

Experiment	**Mean**		**Std Dev**		**Shapiro-Wilk**		**t-test**	**U-test**
	Compliant	Violating	Compliant	Violating	Compliant	Violating	p	p
Use lazy attribute	11.624	11.494	0.330	0.400	0.760	0.223	0.194	
Preload link	11.102	11.240	0.298	0.276	0.454	0.561	0.087	
Avoid using canvas	11.448	11.081	0.276	0.430	0.560	0.554	< 0.001	
Avoid using GIF	44.546	42.167	0.655	0.694	0.572	0.489	< 0.001	
Use standard fonts	11.609	11.699	0.230	0.292	0.547	0.819	0.228	
Avoid logging often	11.344	11.360	0.422	0.247	0.126	0.610	0.867	
Use document fragment	77.685	77.332	2.473	2.274	0.387	0.14	0.566	
Use intersection observer	11.453	11.358	0.312	0.295	0.885	0.693	0.256	
Use request animation frame	35.657	37.548	0.834	2.062	0.63	0.033		< 0.001
Avoid expensive identifiers	96.843	160.024	1.696	3.127	0.880	0.625	< 0.001	
Avoid resizing images	11.691	11.641	0.250	0.331	0.623	0.297	0.533	
Respecting the BF-cache	289.987	291.696	6.926	6.756	0.487	0.856	0.337	

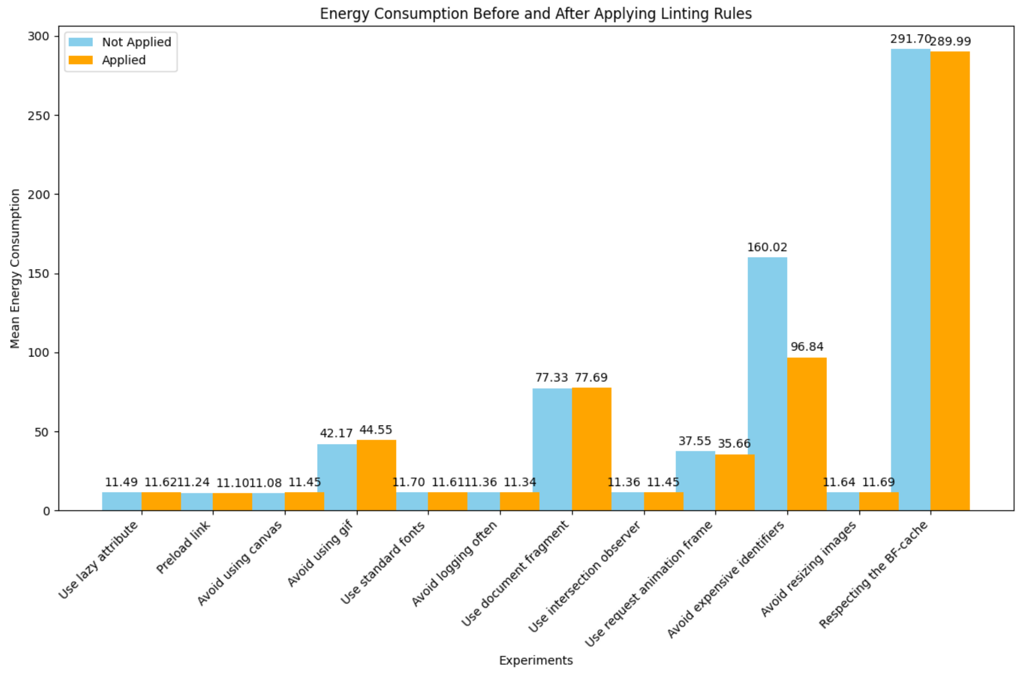


Fig. 1. Energy consumption comparison between the unoptimized and optimized rules (Joules)

API calls to the backend upon loading. For these scenarios, the results for the optimal pattern might have been significantly better, as the responses for those calls would be cached within the page.

Finally, 'avoid using canvas' and 'avoid using GIF' revealed that the supposed anti-pattern was, in fact, more energy efficient. For possible justifications, modern browsers often offload rendering tasks like canvas drawing to the GPU, which we did not account for in our energy measurements [39]. The GPU rendering might also make it more efficient than using DOM-based animations. As for GIFs, since they are generally less compressed, this might reduce the computational effort required for decoding during rendering, leading to less consumption.

Implications. This study has implications for developers working on web development, browser engines, or linters, as our findings help improve coding practices, tool efficiency, and overall software quality.

The results provide insights into energy-inefficient coding anti-patterns and their potential optimizations. These findings can inform updates to coding guidelines and industry standards, helping developers write cleaner, more energy-efficient code that better aligns with best practices. In addition, our study motivates that these idioms can be automatically detected and enforced by linters, reducing inefficiencies at scale. Understanding these patterns may also allow browser engines to mitigate the impact of energy-inefficient code, or even allow language developers to replace and deprecate the

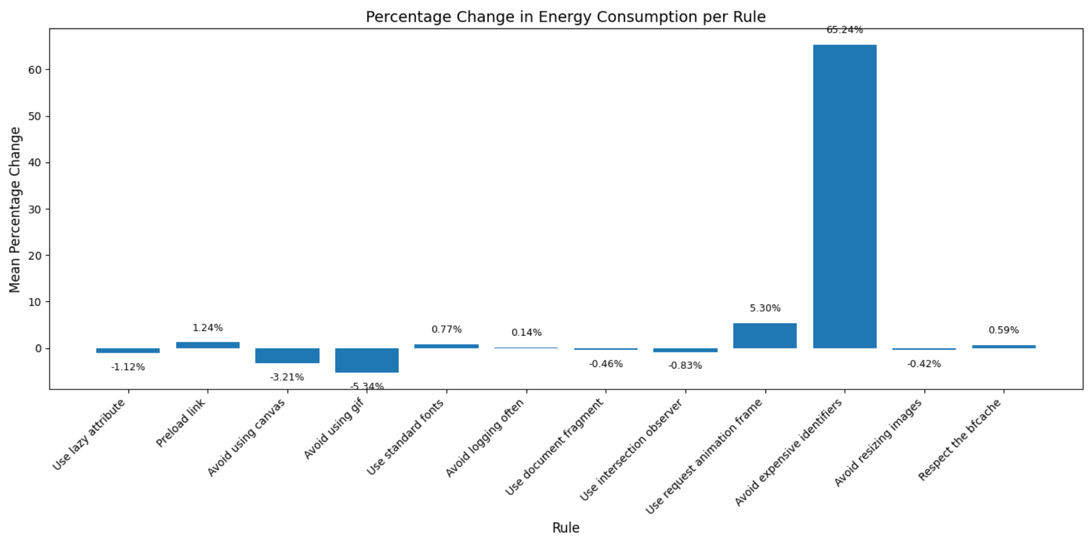


Fig. 2. Percentage change in energy consumption (Joules)

TABLE II
STATISTICAL ANALYSIS OF ENERGY CONSUMPTION PER RULE

Experiment	**Mean diff** (ΔX)	**Mean change** (%)	**Effect size (Cohen's d)**
Use lazy attribute	-0.130	-1.117%	-0.354
Preload link	0.138	1.239%	0.480
Avoid using canvas	-0.368	-3.213%	-1.019
Avoid using GIF	-2.379	-5.34%	-3.526
Use standard fonts	0.089	0.767%	0.339
Avoid logging often	0.016	0.139%	0.046
Use document fragment	-0.354	-0.456%	-0.149
Use intersection observer	-0.095	-0.829%	-0.313
Use request animation frame	1.891	5.303%	N/A
Avoid expensive identifiers	63.180	65.240%	25.115
Avoid resizing images	-0.050	-0.424%	-0.169
Respect the bfcache	1.709	0.589%	0.25

most problematic coding constructs.

Limitations. While our study provides insights into energy consumption on the Web, we must also acknowledge certain limitations that may affect the interpretation of the results.

First, the scope of the experiment is relatively small, mainly due to time constraints. Having only 12 rules is a good starting point for analyzing bad practices in web development, but it is not representative of the full range of energy-intensive behaviors on the web. It is also worth noting that the experiment was conducted in just one language (JavaScript), using one specific browser (Chromium) and one specific linter (ESLint). A broader investigation that includes different browser engines, frameworks, and linters would strengthen the generalizability of the results.

Second, another important consideration is that the experiments focus solely on measuring the energy consumption of JavaScript execution in the browser. Other energy-intensive factors, such as the cost of transmitting, decoding, and caching media assets like GIFs and WEBP, are not considered. Network-related energy consumption is outside the scope of this study.

Third, the examples used in the experiments are not representative of code found in production environments. The test instances consist of isolated patterns within an HTML page, which may not fully reflect how these patterns are typically integrated into real-world websites. As a result, there may be differences in energy consumption when these elements interact with other components, scripts, or user behavior. To obtain more realistic results, future experiments should be conducted in real-world projects to evaluate the effectiveness of our findings in practical scenarios.

Furthermore, while the code used in these experiments is functional and provides meaningful insights, it is likely not optimal. There may be opportunities for efficiency and correctness improvements within the codebase. In addition, the implementation of the experiment is relatively simple, primarily opening a browser and, in the case of the bfcache rule, navigating away and back. This functionality can be extended to account for more complex behavior, allowing for a more comprehensive analysis of energy consumption in more diverse scenarios.

Finally, it is important to recognize that this work is only concerned with energy consumption; there is more to consider. For example, lazy loading of some web elements may indeed reduce energy consumption. However, according to a Cloudflare report, lazy loading can result in multiple server requests as users navigate the page, potentially adding overhead and impacting performance [40]. Therefore, the choice between implementation and energy consumption depends on the specific requirements of a project. It is also worth noting that there is a trade-off between efficiency/performance and maintainability. In some cases, the most energy-efficient solution may result in code that is harder to understand, modify, or debug. So, developers need to consider this for their codebases. Our goal is to educate users about potential energy hotspots so that they can make informed decisions based on their own needs.

VI. Conclusion and Future Work

In conclusion, our work demonstrates that incorporating energy awareness into standard web development practices can yield benefits in sustainable software engineering, thereby reducing the environmental impact of software. By focusing on JavaScript and extending ESLint with custom rules, we have identified how common coding patterns affect energy consumption in measurable ways. While some optimizations (such as caching expensive identifiers) led to striking improvements in overall energy consumption, other presumed ”green” patterns had minimal or even counterintuitive effects.

These findings highlight the complexity of energy-aware development: performance optimizations do not always translate into lower energy consumption, and some best practices (e.g., avoiding GIFs altogether) may involve tradeoffs in terms of feature set, user experience, or code maintainability. Although our experiments focused only on Chromium under controlled conditions, they illustrate the value of tool-assisted, empirical investigations into software sustainability.

Future Work. Looking ahead, this research opens up several avenues for future work. The most valuable direction would be to integrate these experiments into real-world projects to validate the effectiveness of the approach in practical web applications rather than isolated test cases.

Similarly, extending the scope beyond JavaScript would be highly valuable. A large quantity of websites today rely on frameworks like jQuery, React, or Angular [41]. These frameworks introduce much more functionality and complexity because their patterns are independent of JavaScript. Thus, identifying energy hotspots in these tools can provide significant insight into optimizing real-world applications.

In addition, future studies could improve the experimental setup by conducting tests in more diverse environments, across different browsers (e.g., Firefox, Safari) and different devices, to assess the consistency of the results and mitigate potential biases. This would help determine whether the energy savings recommendations generalize across different platforms or require browser-specific optimizations.

Another natural extension would be to expand the set of rules evaluated to include a wider range of energy consumption patterns. Analyzing more complex functionality would provide a more comprehensive analysis of consumption across the web.

Finally, a user study could be conducted to assess whether developers find our linting tool useful and to get feedback on possible ways to improve the tool. Such a study would be crucial for understanding real-world usability, including whether the tool’s energy-saving rules introduce unintended trade-offs, such as reduced code readability, maintainability, or even runtime performance. Understanding these factors is essential, as we hope this study encourages the push towards integrating energy-aware linting tools into existing continuous integration pipelines.

References

[1] H. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, “Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications,” 11 2010, pp. 421–426.

[2] P. Rani, J. Zellweger, V. Kousadianos, L. Cruz, T. Kehrer, and A. Bacchelli, “Energy patterns for web: An exploratory study,” in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*. IEEE, 2024, pp. 12–22. [Online]. Available: https://arxiv.org/abs/2401.06482

[3] B. Couriol, “Co2.js helps developers track their application’s carbon footprint,” InfoQ, 05 2024. [Online]. Available: https://www.infoq.com/news/2024/05/co2-js-carbon-footprint-release

[4] “Stack overflow developer survey,” 2024, accessed: 19 March 2025. [Online]. Available: https://survey.stackoverflow.co/2024/technology#top-paying-technologies

[5] M. Selakovic and M. Pradel, “Performance issues and optimizations in javascript: An empirical study,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 61–72.

[6] T. Imura, “10 recommendations for green software development,” *Green Software Foundation*, 11 2021. [Online]. Available: https://greensoftware.foundation/articles/10-recommendations-for-green-software-development

[7] R. D. Caballar, “We need to decarbonize software: The way we write software has unappreciated environmental impacts,” *IEEE Spectrum*, vol. 61, no. 4, pp. 26–31, 2024.

[8] K. Carter, S. M. G. Ho, M. M. A. Larsen, M. Sundman, and M. H. Kirkeby, “Energy and time complexity for sorting algorithms in java,” 2024. [Online]. Available: https://arxiv.org/abs/2311.07298

[9] J. Pallister, S. J. Hollis, and J. Bennett, “Identifying compiler options to minimize energy consumption for embedded platforms,” *The Computer Journal*, vol. 58, no. 1, p. 95–109, Nov. 2013. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxt129

[10] S. Hu and L. K. John, “Impact of virtual execution environments on processor energy consumption and hardware adaptation,” in *Proceedings of the 2nd International Conference on Virtual Execution Environments*, ser. VEE ’06. New York, NY, USA: Association for Computing Machinery, 2006, p. 100–110. [Online]. Available: https://doi.org/10.1145/1134760.1134775

[11] T.-J. Yang, Y.-H. Chen, J. Emer, and V. Sze, “A method to estimate the energy consumption of deep neural networks,” in *2017 51st Asilomar Conference on Signals, Systems, and Computers*, 2017, pp. 1916–1920.

[12] N. C. Zakas, “Eslint: Pluggable javascript linter,” https://eslint.org, 2013, accessed: April 4, 2025.

[13] A. Kovalyov, “Jshint: A javascript code quality tool,” https://jshint.com, 2011, accessed: April 4, 2025.

[14] C. Brosch, “Influence of static code analysis on energy consumption of software,” in *EnviroInfo 2023: Environmental Informatics*. Springer, 2023, pp. 1–12.

[15] A. A. Bangash, K. Eng, Q. Jamal, K. Ali, and A. Hindle, “Energy consumption estimation of api-usage in smartphone apps via static analysis,” in *20th International Conference on Mining Software Repositories (MSR)*. IEEE/ACM, 2023, pp. 272–283. [Online]. Available: https://doi.org/10.1109/MSR59073.2023.00042

[16] L. Li, Y. Liu, Y. Liu, D. Lo, L. Jiang, and L. Zhang, “Assisting developers perform empirical study on energy consumption of mobile applications,” in *2020 IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2020, pp. 13–24.

[17] S. Wegener, K. K. Nikov, J. Nunez-Yanez, and K. Eder, “Energyanalyzer: Using static wcet analysis techniques to estimate the energy consumption of embedded applications,” in *21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023)*, ser. Open Access Series in Informatics (OASIcs), vol. 114. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023, pp. 9:1–9:14. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2023.9

[18] D. Sousa, F. Medeiros, M. Ribeiro, V. Alves, P. Borba, N. Siegmund, and J. Guo, “Statically analyzing the energy efficiency of software product lines,” *Journal of Software Engineering Research and Development*, vol. 6, no. 1, pp. 1–18, 2018.

[19] N. Grech, K. Georgiou, S. Kerrison, Z. Wang, and K. Eder, “Static analysis of energy consumption for llvm ir programs,” in *Proceedings*

of the 18th International Workshop on Software and Compilers for Embedded Systems (SCOPES). ACM, 2015, pp. 12–21. [Online]. Available: https://doi.org/10.1145/2764967.2764974

[20] Y. Huizhan, C. Juan, and Y. Xuejun, "Static wcet analysis based compiler-directed dvs energy optimization in real-time applications," in *International Conference on Advances in Computer Systems Architecture*. Springer, 2006, pp. 108–121.

[21] C. C. N. Responsable, "115 web ecodesign best practices," 2024, accessed: 2025-04-03. [Online]. Available: https://collectif.greenit.fr/ecoconception-web/

[22] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: how do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031

[23] Green-Code-Initiative, "Creedengo-javascript/eslint-plugin/readme.md at main · green-code-initiative/creedengo-javascript," Mar 2025. [Online]. Available: https://github.com/green-code-initiative/creedengo-javascript/blob/main/eslint-plugin/README.md

[24] Mozilla, *Lazy Loading - Web Performance*, MDN Web Docs, 2025, accessed: 2025-04-03. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Performance/Guides/Lazy_loading

[25] ——, *rel=preload - HTML: Hypertext Markup Language*, MDN Web Docs, Mar. 2025, accessed: 2025-04-03. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/rel/preload

[26] B. Poulain and S. Fraser, "How web content can affect power usage," August 27 2019. [Online]. Available: https://webkit.org/blog/8970/how-web-content-can-affect-power-usage/

[27] P. Garaizar, M. A. Vadillo, and D. L. de Ipiña, "Presentation accuracy of the web revisited: Animation methods in the HTML5 era," *PLoS ONE*, vol. 9, no. 10, p. e109812, 2014. [Online]. Available: https://doi.org/10.1371/journal.pone.0109812

[28] CNUMR, *115 Web Ecodesign Best Practices*, GitHub, 2025, accessed: 2025-04-03. [Online]. Available: https://github.com/cnumr/best-practices

[29] B. Dornauer, W. Vigl, and M. Felderer, "On the role of font formats in building efficient web applications," 2023. [Online]. Available: https://arxiv.org/abs/2310.06939

[30] MDN Web Docs contributors, *Main Thread*, Mozilla, Dec. 2024, accessed: 2025-04-03. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Main_thread

[31] Mozilla, *Document: createDocumentFragment() Method - Web APIs*, MDN Web Docs, Mar. 2024, accessed: 2025-04-03. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document/createDocumentFragment

[32] MDN Web Docs contributors, *JavaScript Performance Optimization*, Mozilla, 2025, accessed: 2025-04-03. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Performance/JavaScript#tips_for_writing_more_efficient_code

[33] MDN Web Docs, *Intersection Observer API*, Mozilla, 2025, accessed: 2025-04-03. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API

[34] MDN Web Docs contributors, *Window: requestAnimationFrame() method*, Mozilla, 2025, accessed: 2025-04-03. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/requestAnimationFrame

[35] ——, *bfcache*, Mozilla, Sep. 2024, accessed: 2025-04-03. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/bfcache

[36] J. Sallou, L. Cruz, and T. Durieux, "Energibridge: Empowering software sustainability through cross-platform energy measurement," *arXiv preprint arXiv:2312.13897*, 2023. [Online]. Available: https://arxiv.org/abs/2312.13897

[37] "What is chromium, and how does it enhance your browser?" Microsoft, 2025. [Online]. Available: https://www.microsoft.com/en-us/edge/learning-center/what-is-chromium-how-does-it-enhance-your-browser?form=MA13I2

[38] N. U. Library, "Cohen's d - statistics resources," accessed: April 3, 2025. [Online]. Available: https://resources.nu.edu/statsresources/cohensd

[39] S. Chikuyonok. (2016, Dec.) Css gpu animation: Doing it right. Accessed: 2025-04-04. [Online]. Available: https://www.smashingmagazine.com/2016/12/gpu-animation-doing-it-right/

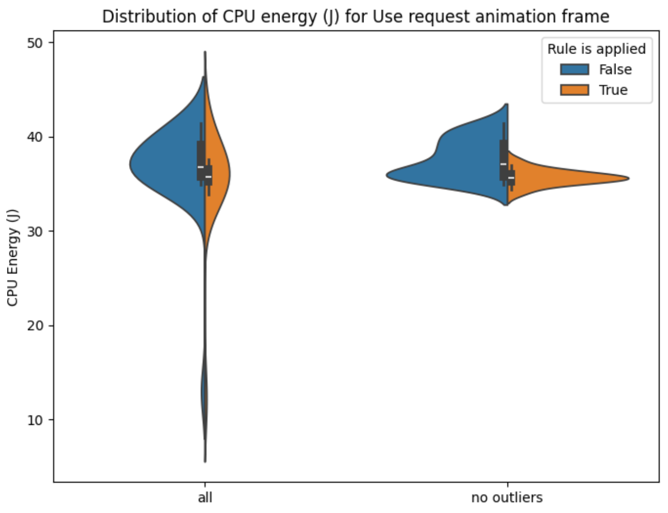
[40] "What is lazy loading?" Cloudflare.com, 2024. [Online]. Available: https://www.cloudflare.com/learning/performance/what-is-lazy-loading/

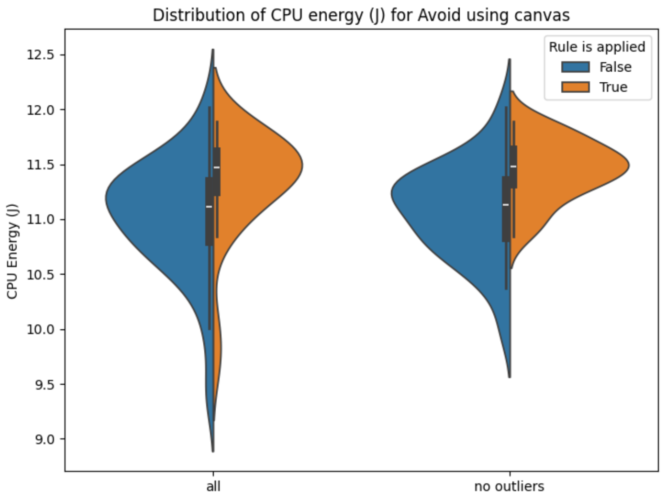
[41] "Stack overflow developer survey," 2024, accessed: 4 April 2025. [Online]. Available: https://survey.stackoverflow.co/2024/technology#1-web-frameworks-and-technologies

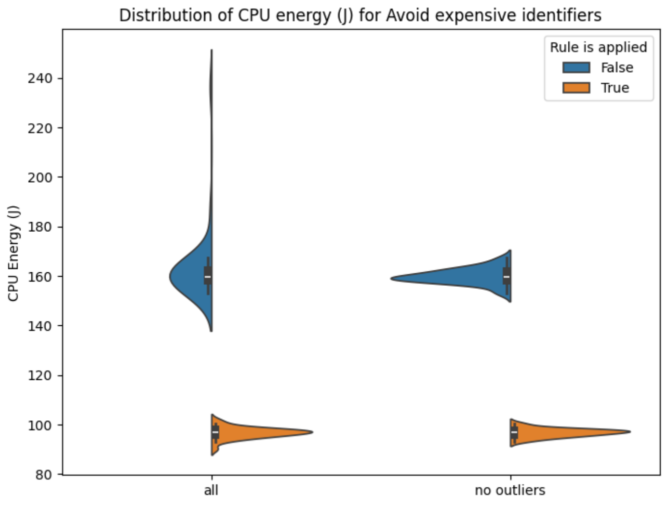
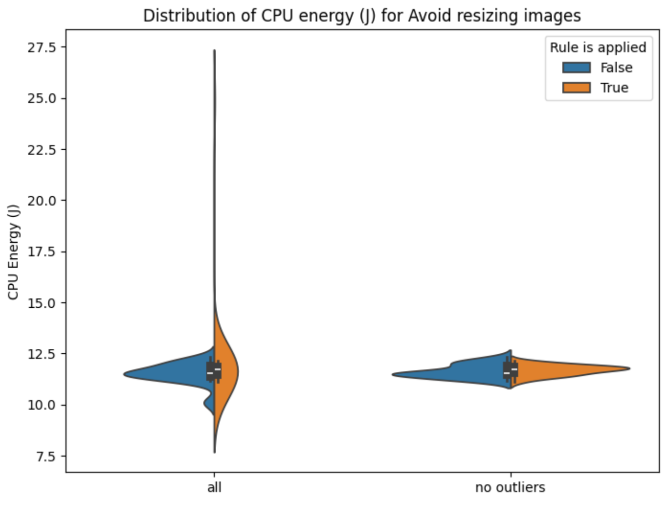
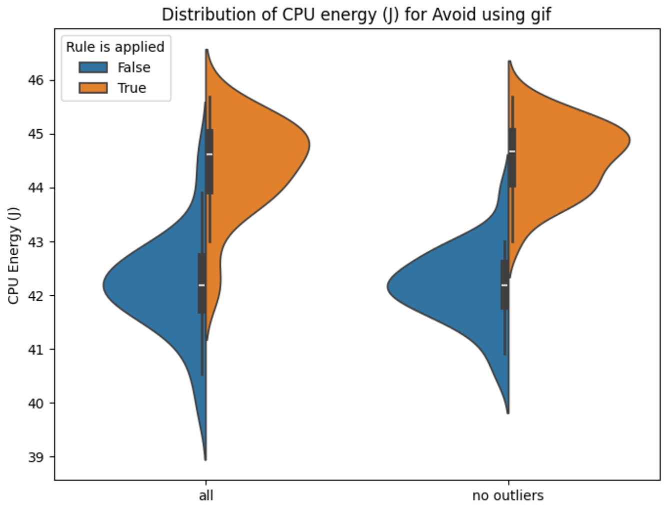
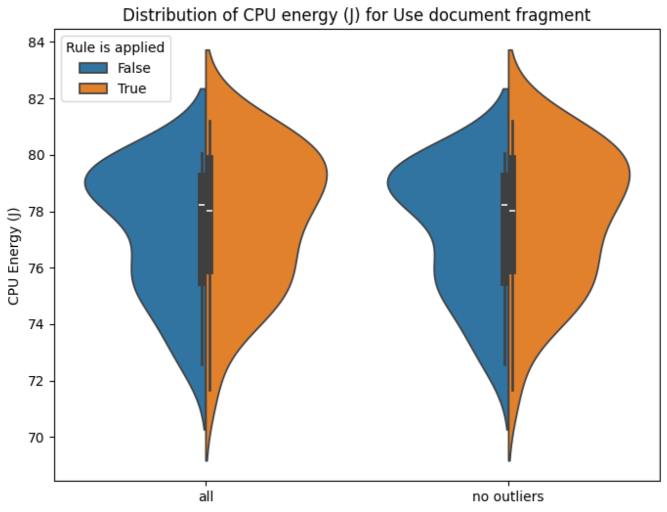
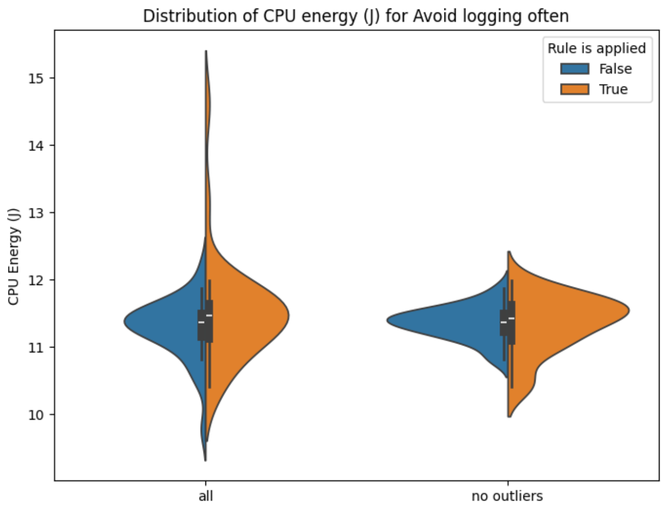
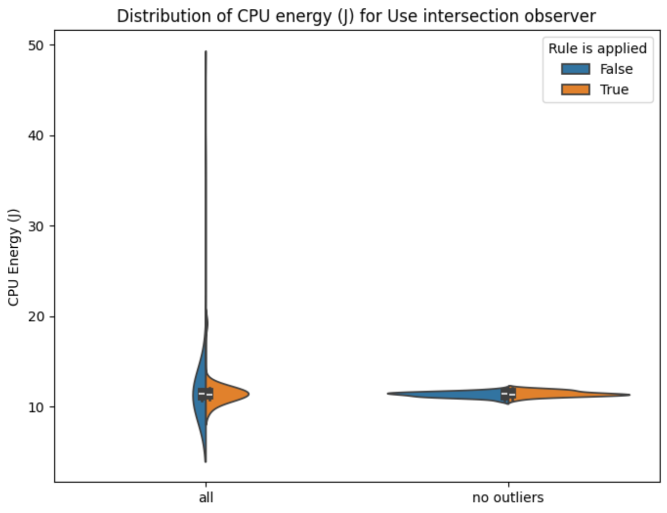
VII. Appendix

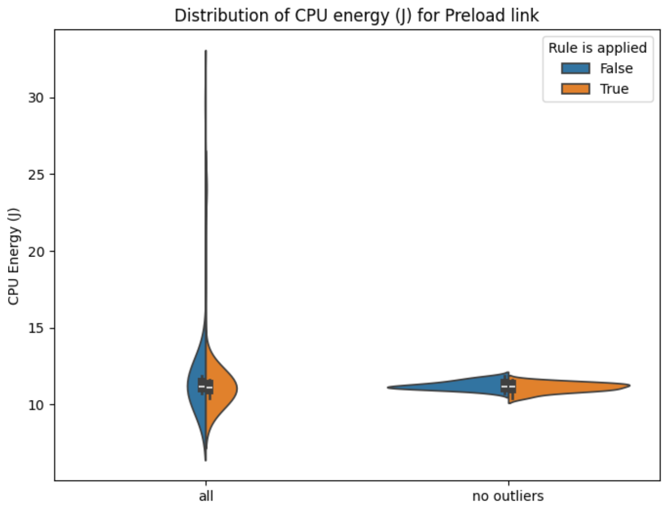
A. Result violin plots

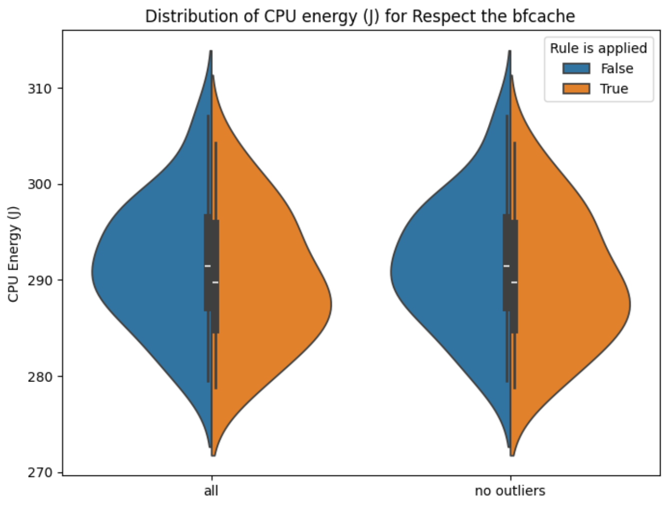
Each plot shows the results for the respective experiment. Note that for "all", each of the samples is included, while for "no outliers", we remove the outliers as described in section III-E.

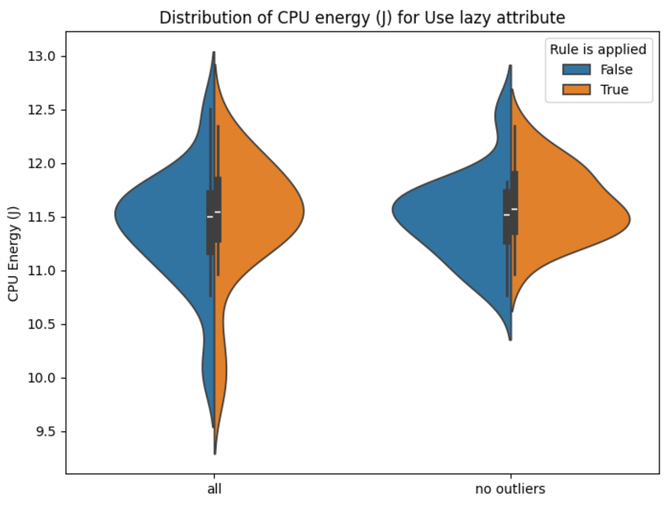


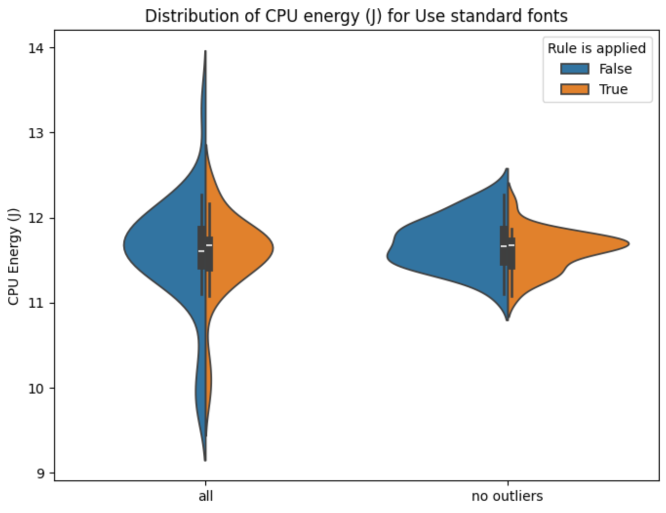



Distribution of CPU energy (J) for Avoid expensive identifiers
Rule is applied
False
True
CPU Energy (J)
all
no outliers

Distribution of CPU energy (J) for Avoid resizing images
Rule is applied
False
True
CPU Energy (J)
all
no outliers

Distribution of CPU energy (J) for Avoid using gif
Rule is applied
False
True
CPU Energy (J)
all
no outliers

Distribution of CPU energy (J) for Use document fragment
Rule is applied
False
True
CPU Energy (J)
all
no outliers

Distribution of CPU energy (J) for Avoid logging often
Rule is applied
False
True
CPU Energy (J)
all
no outliers

Distribution of CPU energy (J) for Use intersection observer
Rule is applied
False
True
CPU Energy (J)
all
no outliers


Distribution of CPU energy (J) for Preload link
Rule is applied
False
True
CPU Energy (J)
10
15
20
25
30
all
no outliers


Distribution of CPU energy (J) for Respect the bfcache
Rule is applied
False
True
CPU Energy (J)
270
280
290
300
310
all
no outliers


Distribution of CPU energy (J) for Use lazy attribute
Rule is applied
False
True
CPU Energy (J)
9.5
10.0
10.5
11.0
11.5
12.0
12.5
13.0
all
no outliers


Distribution of CPU energy (J) for Use standard fonts
Rule is applied
False
True
CPU Energy (J)
9
10
11
12
13
14
all
no outliers