Comparing the Environmental Footprint of Web Stacks in the RealWorld Apps

Allysson Guimarães
Universidade Federal do Agreste de Pernambuco
Garanhuns, Brazil
allysson.guimaraes@ufape.edu.br

Rodrigo Andrade
Universidade Federal do Agreste de Pernambuco
Garanhuns, Brazil
rodrigo.andrade@ufape.edu.br

ABSTRACT

As concerns about the environmental impact of software continue to grow, recent studies have explored how software consumes energy and emits greenhouse gases, often using isolated benchmarks or specific tasks. However, the environmental implications of implementing the same application across different technology stacks remain underexplored. In this paper, we investigate the energy consumption, execution time, and carbon dioxide emissions of a benchmark web application implemented using three distinct technology stacks. Through 32 usage scenarios and the analysis of 10 efficiency-related metrics, our results reveal no statistically significant differences among the stacks regarding their environmental footprint. These findings suggest that replacing the entire technology stack may not be as impactful as optimizing specific application functionalities when aiming for more sustainable software.

KEYWORDS

Web application technology stacks, energy consumption, execution time, carbon dioxide emission

1 Introduction

Recently, efforts to reduce the environmental footprint of software have garnered growing attention [24, 31, 37, 56]. This rising interest highlights the need to optimize both software development and usage to mitigate environmental harm, particularly by enhancing the efficiency of computational resource consumption, such as energy usage [49]. Within this context, Information and Communication Technologies have been estimated to contribute between 1.8% and 3.9% of global Greenhouse Gas emissions (GHG) [24], such as carbon dioxide. Alarmingly, these figures are projected to rise in the coming years [31].

In pursuit of more environmentally sustainable software development and usage, researchers have sought to understand how software consumes energy [50, 57, 60] and the extent of its GHG emissions [5, 11]. Studies on energy consumption often rely on benchmarks consisting of isolated algorithms implemented in various programming languages [57]. In the case of GHG emissions, estimations typically focus on the amount of carbon dioxide produced during the execution of specific tasks, such as training large language models [5].

While existing approaches are valuable for understanding how to reduce energy consumption and GHG emissions, they often overlook the impact that different implementations of the same application can have on these factors. For instance, how do choices regarding programming languages, tools, and frameworks influence the environmental impact of an application's execution? To the best of our knowledge, no prior study has investigated whether developers should consider selecting or replacing these development approaches as a strategy to reduce long-term energy consumption and carbon dioxide emissions.

To address part of this gap, our study investigates the energy consumption, execution time, and carbon dioxide emissions of a single web application implemented using three distinct technology stacks (e.g., Angular with Django). For this purpose, we leverage an existing benchmark web application, namely RealWorld App, that provides multiple versions built with different stacks. Based on this application, we define 32 usage scenarios to carry out our evaluation. In this context, we analyze energy consumption, execution time, and carbon dioxide emissions across these 32 testing scenarios implemented in different web application technology stacks.

To support our assessment, we select a set of 10 metrics to evaluate the energy, time, and carbon efficiency of each technology stack. Interestingly, our findings suggest that the choice of stack has minimal impact on the overall environmental footprint. For the same testing scenarios implemented across the three stacks, variations in energy consumption, execution time, and CO_2 emissions remain below 3%. Moreover, we observe no statistically significant differences among the stacks with respect to any of these factors. These results point to the need for further investigation into whether optimizing individual application functionalities may offer greater environmental benefits than replacing the entire technology stack.

The remainder of this work is organized as follows. Section 2 explains the adopted structure and methodology. Section 3 details the results for energy consumption, execution time, and carbon dioxide emissions. Moreover, we discuss related work in Section 4 and conclude this study in Section 5.

2 Study Settings

This section outlines the structure and methodology of our study. We begin by detailing our goal, research questions, and metrics in Section 2.1. Section 2.2 presents the set of applications used as case studies, along with their respective technology stacks. Moreover, Section 2.3 describes the tool we use to create the testing scenarios for our measurements. Next, in Section 2.4, we introduce the tool selected for performing the measurements. Finally, Section 2.5 provides information on our execution environment.

2.1 Goal-Question-Metric

To better drive our work, we use the Goal-Question-Metric (GQM) framework [7]. Table 1 summarizes our approach.

To achieve our goal of analyzing energy efficiency in relation to the environmental impact of computing systems, we define three research questions. First, **RQ1** investigates how the execution of tasks across different technologies influences energy consumption. For

Table 1: Our Goal-Question-Metric approach

Goal	
Purpose	To analyze the energy efficiency of web applications
Issue	in relation to their environmental impact
Object	across various testing scenarios implemented in different web application technology stacks for the RealWorld App
Viewpoint	from a user's perspective
Questions	**Metrics**

RQ1- How energy-efficient are different web application technology stacks in terms of energy consumption?	`-Mean in Watts (MnW)`
	`-Median in Watts (MdW)`
	`-Total Consumption in Watts (TC)`
	`-Watt-hour per Second (Wh/s)`

RQ2- How time-efficient are different web application technology stacks?	`-Total Execution Time in Seconds (TE)`
	`-Mean Execution Time in Seconds (ME)`
	`-Median Execution Time in Seconds (MD)`

RQ3- How carbon-efficient are different web application technology stacks?	-Total CO_2 Emissions in Grams (TCO_2) -CO_2 Equivalent Emissions in Grams per Second (CO_2eq/s) -Estimated CO_2 Equivalent Emissions in Grams per Year (CO_2eq/y)

RQ4- Are the differences in energy consumption, execution time, and CO_2 emissions among technology stacks statistically significant enough to be a primary factor in their selection?	`ANOVA p-value`
	`Effect Size (Cohen's d)`
	`Pearson Correlation Coefficient`

instance, does the execution of a login feature implemented using React [55] and Rails [46] consume more energy than the execution of the same feature implemented in Angular [4] and Django [52]? An answer to this question could assist developers in selecting the technology stack that best meets their energy efficiency requirements.

To answer **RQ1**, we select four metrics, as shown in Table 1. The `MnW` metric represents the mean power consumption, measured in watts, during the execution of a system's feature, repeated ten times, implemented in a specific technology stack. The `MdW` metric measures the median instead of the mean for the same power consumption. Furthermore, the `TC` metric sums the watts consumed by the 10 executions of each feature. At last, `Wh/s` indicates the rate of energy consumption over time, that is, how many watt-hours each feature consumes per second.

Additionally, we define **RQ2** to assess the performance of task execution across different technology stacks. For instance, does logging out in an application built with React and Rails execute faster than the same feature implemented in KVision [39] and Spring Boot [9]? By addressing this research question, we aim to help developers make more informed decisions when selecting the most suitable technology stack for their software projects.

In this context, we define two metrics to answer **RQ2** (Table 1). The `TE` metric measures the total execution time, in seconds, for all repeated runs of a given feature. Additionally, the `ME` metric represents the average execution time per run, calculated by dividing `TE` by the number of executions for the measured feature.

Furthermore, our objective with **RQ3** is to quantify CO_2 emissions, given their significant environmental impact [6, 31, 40]. Notably, the Information and Communication Technology (ICT) sector alone is estimated to contribute between 1.8% and 2.8% of global Greenhouse Gas (GHG) emissions [24]. In this context, software execution is an integral part of ICT, and CO_2 is among the primary GHGs [41]. Therefore, answering **RQ3** might help developers decide which technology stack contributes to reducing CO_2 emissions.

To answer **RQ3**, we select three metrics. First, TCO_2 measures the total CO_2 emissions (in grams) across all repeated executions of a given feature. Second, CO_2eq/s represents the carbon dioxide equivalent emissions (in grams per second) [23]. This metric accounts for not only CO_2 but also other greenhouse gases, such as methane and nitrous oxide, by expressing their impact in terms of CO_2 equivalence. Therefore, CO_2eq/s quantifies the combined effect of multiple greenhouse gases in a standardized unit. Finally, we include the CO_2eq/y metric, which estimates the total CO_2 equivalent emissions (in grams) over the course of a year.

At last, **RQ4** investigates whether the choice of a technology stack to implement the RealWorld App significantly influences energy consumption, execution time, or carbon dioxide emissions. Therefore, we assess whether the technology stack choice should be a primary factor for reducing or increasing such measures.

To answer **RQ4**, we use three statistical methods to investigate differences among the selected technology stacks: (i) analysis of variance (ANOVA) [59], (ii) effect size calculations using Cohen's d[14], and (iii) Pearson correlation test [8].

2.2 RealWorld apps

As a case study, we use the RealWorld App [65], a blog platform inspired by Medium [42], developed as a benchmark for web application development. Unlike typical demo applications, it includes a set of features commonly found in real-world web systems, such as user authentication, login, and profile management, full CRUD functionality for creating and editing articles, an integrated commenting system, global and personalized feeds, and the ability to follow other users.

In this context, to carry out our study, we select three technology stacks provided by the RealWorld App: (i) React with Rails, (ii) Angular with Django, and (iii) KVision with Spring Boot. We base this selection on our familiarity with the technologies, their popularity among software developers [34], and the requirement that the corresponding repositories have been updated within the last three years.

To systematically select usage scenarios for the RealWorld App, we utilize the E2E Code Coverage tool [16]. Our selection criterion is to achieve a minimum of 80% code coverage for source code statements, functions, and lines. This process results in a final suite of 32 testing scenarios that exercise the application's existing features. Table 2 presents the consolidated E2E Code Coverage report for this suite. For instance, the 32 scenarios collectively execute 84.7% of all statements in the RealWorld App's source code.

Table 2: E2E Code Coverage Report

	Statements	**Functions**	**Lines**
Coverage percentage	84.7%	80.98%	85.5%

Additionally, we categorize these 32 scenarios into two groups: *Simple* and *Composed*, as shown in Tables 3 and 4, respectively. A *Simple* scenario involves the execution of a single feature. For example, the `createArticle` scenario focuses solely on creating a new article, without invoking any additional functionality. In contrast, a *Composed* scenario consists of at least two sequential feature executions. For instance, the `loginAndLogout` scenario evaluates a user logging in and subsequently logging out. This categorization helps to better assess the energy efficiency of different technology stacks under varying usage patterns.

Table 3: RealWorld App *Simple* testing scenarios

Testing scenario	**Description**
`login`	Logging in existing user
`signUp`	Creating user
`logout`	Logging out existing user
`invalidEmail`	Logging in with an invalid email
`invalidPassword`	Login with an invalid password
`updateProfile`	Updating user profile
`createArticle`	Creating an article
`deleteArticle`	Deleting existing article
`updateArticle`	Updating article
`createComment`	Commenting an existing article
`deleteComment`	Deleting an existing comment
`filterTag`	Filtering article by tag
`loadPages`	Loading pages
`loadPageAndWait`	Loading a page

Table 4: RealWorld App *Composed* testing scenarios

Testing scenario	**Description**
`signUpAndLogin`	Creating user and logging in
`loginAndLogout`	Logging in and out an existing user
`createAndDeleteArticle`	Creating deleting an article
`createAndDeleteComment`	Creating and deleting a comment
`favoriteAndUnfavoriteArticle`	Favoriting and unfavoriting an article
`loginAndCreateArticle`	Logging in and creating an article
`loginAndDeleteArticle`	Logging in and deleting an article
`loginCreateAndDeleteArticle`	Logging in, creating, and deleting an article
`loginCreateComment`	Logging in and commenting on an article
`loginAndDeleteComment`	Logging in and deleting a comment
`loginCreateAndDeleteComment`	Logging in, creating, and deleting a comment
`loginAndUpdateProfile`	Logging in and updating an user profile
`loginCreateArticleAndComment`	Logging in, creating an article, and commenting
`signUpLoginAndLogout`	Creating user, logging in and out
`loginFilterTagAndComment`	Logging in, filtering articles, and commenting
`loginCreateAndUpdateArticle`	Logging in, creating, and updating an article
`createAndUpdateArticle`	Creating and updating an article
`createAndCommentArticle`	Creating an article and commenting

In this context, we execute the same testing scenarios across different technology stacks. For example, we compare the carbon dioxide emissions produced when logging in an existing user in a React with Rails implementation versus an Angular with Django implementation. We repeat each execution 10 times.

2.3 Cypress

To execute the set of selected tasks in our case study, we employ the Cypress testing framework [17]. Cypress is an open-source end-to-end testing tool specifically designed for web applications and is agnostic to the underlying technology stack. We use this framework to implement automated tests for each scenario described in Tables 3 and 4, covering all three technology stacks: React with Rails, Angular with Django, and KVision with Spring Boot. As an example, Listing 1 shows the implementation of the `deleteArticle` testing scenario for the React with Rails stack.

This Cypress test simulates the deletion of an article by an authenticated user. First, in lines 3 and 4, a JWT token is stored in the browser's localStorage to simulate a logged-in session. The test scenario then begins by navigating to the profile page of a user named "user1" (line 7). It clicks on the first article preview link to open the article page in line 8. A helper function is defined to retrieve the article's title element in line 8, which will later be used for validation. Then, it locates and clicks the "Delete Article" button in line 10. Finally, it asserts that the title element no longer exists on the page, confirming that the article was successfully deleted in line 11. To gather measurements for the metrics presented in Table 1, we execute each Cypress test, including the example in Listing 1, ten times.

Listing 1: `deleteArticle` testing scenario for React with Rails

```
1  describe('test article delete', () => {
2   before(() => {
3    const jwtToken = 'tokenSecret';
4    window.localStorage.setItem('jwtToken', jwtToken);
5  });
6   it('delete article', () => {
7    cy.visit('http://localhost:4000/#/profile/user1');
8    cy.get('.preview-link').first().click();
9    const title = () => cy.get('h1');
10   cy.get('button').contains(' Delete Article').click();
11    title().should("not.exist");
12  });
13 });
```

2.4 Code Carbon

To automate the measurement of our selected metrics, we use the CodeCarbon tool [11, 15]. This open-source tool is designed to estimate and track the energy consumption, performance, and carbon dioxide emissions of software execution. It also features automatic logging, enabling detailed analysis of energy usage. Additionally, CodeCarbon converts electricity consumption into CO_2 equivalent (CO_2eq) emissions, incorporating regional energy mix data to provide environmental impact assessments [38].

In this context, to estimate CO_2 emissions, we configure CodeCarbon with the default setting for Brazil, which assumes a carbon intensity of 98.348 grams of CO_2 equivalent per kilowatt-hour (gCO_2eq/kWh) [47]. This value indicates that, on average, the Brazilian energy mix emits 98.348 grams of greenhouse gases (in CO_2 equivalent) for every kilowatt-hour of electricity produced. The most recent data available refer to the year 2023.

Moreover, CodeCarbon is designed to measure the total CPU usage over a given period of time. However, in our case, we need to track a specific process at a time, each corresponding to a task executed within our case study (Section 2.2). To achieve this, we create a script that monitors the processes running when the Cypress testing scenario is being executed. Essentially, this script automatically starts the task and logs the measurements provided by the CodeCarbon tool.

Listing 2 illustrates part of our script output when using CodeCarbon regarding the execution of `CreateAndCommentArticle` testing scenario (Table 4. This output presents the results of an automated test executed using Cypress, with metrics collected by the CodeCarbon library. Lines 3-7 list the active system processes at the time of execution, which helps contextualize system load and potential interferences. Moreover, line 10 indicates the specific test being run—in this case, the `CreateAndCommentArticle` testing scenario using the Angular with Django stack. Lines 13 and 14 report the carbon emissions and total execution time. Finally, lines 17-19 provide energy usage estimations for each hardware component: RAM showed 2.86 watts consumption, the GPU was not used, and the CPU was the primary source of energy usage, averaging 21.4 watts.

Given the 32 testing scenarios outlined in Tables 3 and 4, and the use of three different technology stacks, we collect a total of 96 script outputs similar to the one shown in Listing 2. We compile these outputs into spreadsheets, which serve as the basis for the results and conclusions presented in Section 3.

Listing 2: Our script output for measuring `CreateAndCommentArticle` with CodeCarbon

```
 1 ...
 2 === Running Processes ===
 3 PID: 1, Name: systemd, User: root
 4 PID: 2, Name: kthreadd, User: root
 5 PID: 3, Name: pool_workqueue_release, User: root
 6 PID: 4, Name: kworker/R-rcu_g, User: root
 7 PID: 5, Name: kworker/R-rcu_p, User: root
 8 ...
 9 === Cypress script ===
10 Script: /angular+django/CreateAndCommentArticle.cy.ts
11
12 === CodeCarbon measures ===
13 Carbon emissions: 0.00011397633492050715 kilograms
14 Execution time: 181.70251083374023 seconds
15
16 === CodeCarbon log ===
17 Energy consumed for RAM : 0.000002 kWh. RAM Power : 2.86 W
18 Energy consumed for all GPUs : 0.000000 kWh. Total GPU Power : 0 W
19 Energy consumed for all CPUs : 0.000012 kWh. Total CPU Power :
       21.40 W
20 ...
```

2.5 Execution Environment

To conduct our study, we use an Acer Nitro 5 laptop equipped with an Intel Core i5-7300HQ processor, 8 GB of RAM, and SSD storage, running Ubuntu 22.04 LTS. To reduce interference from background activities, we disable system features such as Bluetooth, system updates, and Wi-Fi throughout the experiments. Prior to each test execution, we restart the system and leave it idle for three minutes to help with post-boot stabilization. Subsequently, we initialize the application corresponding to the target technology stack. Finally, we execute our script and collect its output for analysis.

Last but not least, we provide every script, setting, and artifact necessary to reproduce this work in our Online Appendix [30].

3 Results and Discussion

This section presents the results of the comparative analysis among the technology stacks React with Rails, Angular with Django, and KVision with Spring Boot. Section 3.1 discusses the energy efficiency results, focusing on CPU power consumption. Section 3.2 examines performance in terms of execution time across the selected testing scenarios, while Section 3.3 presents the corresponding carbon dioxide emissions. Section 3.4 explains the results regarding statistical differences among the technology stacks. In Section 3.5, we provide an integrated discussion of the findings, and in Section 3.6, we outline the threats to the validity of this study.

3.1 Energy efficiency

This section aims to answer **RQ1**. Therefore, we investigate the energy efficiency of the three selected technology stacks in terms of CPU energy consumption. To help us answer this question, we use the Mean in Watts (`MnW`), Median in Watts (`MdW`), Total Consumption in Watts (`TC`), and Watt-hour per second (`Wh/s`) metrics, as explained in Section 2.1. Table 5 shows the results for each metric.

Table 5: Energy results for each technology stack

Stack	React with Rails	Angular with Django	KVision with Spring Boot
`MnW`	22.2311	21.9652	22.3994
`MdW`	23.0995	22.9814	23.1751
`TC`	802.7107	794.4133	808.6571
`Wh/s`	0.22297	0.22067	0.22462

Although the differences are relatively small, Angular with Django exhibits the lowest average energy consumption among the three stacks. Specifically, its consumption of 21.9652 watts is 1.21% lower than that of React with Rails and 1.98% lower than KVision with Spring Boot. However, these results solely are not enough to draw conclusions about energy efficiency. Therefore, we also measure the Total Consumption in Watts (`TC`) for each *Simple* and *Composed* testing scenario (Tables 6 and 7).

As presented in Table 6, the `TC` rates are relatively similar, with differences remaining below 5% (**Diff%**). Total energy consumption per scenario ranges from 17.70 watts for Angular with Django (`loadPagesAndWait`) to 27.48 watts for React with Rails (`deleteArticle`). In this context, Angular with Django demonstrates the best energy efficiency in 8 out of the 14 evaluated scenarios. React with Rails leads in 5 scenarios, while KVision with Spring Boot achieves the best result in only one case (`loadPages`).

Notably, in the `loadPages` scenario, React with Rails consumes 6.21% more energy than KVision with Spring Boot. However, this stack is significantly less efficient in specific scenarios such as `updateProfile` (5.37%) and `filterTag` (4.38%). Furthermore, deletion scenarios exhibit the smallest energy consumption variation among the three technology stacks, with differences remaining below 0.31%, as shown in the three **Diff%** columns.

Furthermore, Table 7 presents the `TC` results for the *Composed* scenarios. Angular with Django stands out as the most energy-efficient stack, achieving the lowest consumption in 10 out of 18 scenarios. However, the overall differences among the stacks remain relatively small, typically under 5% - equal to the *Simple* scenarios.

Table 6: Total Energy Consumption in Watts for *Simple* testing scenarios

Testing Scenario	React with Rails	Diff%	Angular with Django	Diff%	KVision with Spring Boot	Diff%
`createArticle`	22.94	1.18%	22.67	0.00%	23.03	1.61%
`createComment`	27.03	2.15%	26.46	0.00%	26.93	1.79%
`deleteArticle`	27.48	0.31%	27.39	0.00%	27.44	0.18%
`deleteComment`	27.21	0.20%	27.16	0.00%	27.19	0.12%
`filterTag`	26.59	2.80%	25.87	0.00%	27.00	4.38%
`loadPages`	27.39	6.31%	26.21	1.72%	25.77	0.00%
`loadPagesAndWait`	18.12	2.38%	17.70	0.00%	18.24	3.06%
`login`	26.51	0.00%	26.78	1.01%	26.84	1.23%
`loginInvalidEmail`	26.08	0.00%	26.32	0.95%	26.66	2.24%
`loginInvalidPassword`	26.55	1.40%	26.18	0.00%	26.62	1.67%
`logOut`	26.46	0.12%	26.43	0.00%	26.81	1.44%
`singUp`	25.25	0.00%	25.46	0.81%	26.30	4.15%
`updateArticle`	25.24	0.00%	26.00	3.02%	25.75	2.02%
`updateProfile`	24.63	0.00%	25.35	2.91%	25.95	5.37%

KVision with Spring Boot, on the other hand, consistently exhibits higher energy usage, consuming up to 8.07% (**Diff%**) more energy compared to the most efficient alternatives. This trend is especially pronounced in creation and update operations, where KVision regularly shows the least favorable energy consumption. React with Rails occupies a middle ground, leading in energy efficiency in 5 scenarios, particularly in tasks such as `createAndCommentArticle` and `createAndUpdateArticle`.

In this context, the most pronounced differences in energy consumption for *Composed* scenarios occur in workflows, such as `loginCreateArticleAndComment`, where Angular with Django outperforms the other stacks by approximately 22%. In contrast, scenarios like `favoriteAndUnfavoriteArticle` exhibit minimal variation among the three stacks, indicating that some functionalities tend to consume similar amounts of energy regardless of the underlying technology.

Last but not least, in response to **RQ1**, we conclude that Angular with Django consistently exhibits the lowest energy consumption across the majority of testing scenarios for the RealWorld App. Nonetheless, the differences between the stacks are relatively modest—typically below 2% in most cases. Although energy efficiency can vary depending on the specific scenario, the overall variations are minimal. This suggests that, while measurable differences in CPU energy consumption exist, the choice of technology stack has only a limited impact on overall energy efficiency.

3.2 Time efficiency

This section presents the results regarding **RQ2**, which focus on the time efficiency of the three evaluated technology stacks. As discussed in Section 2.1, we use the Total Execution Time in Seconds (`TE`), Mean Execution Time in Seconds (`ME`), and Median Execution Time in Seconds (`MD`) metrics. Table 8 illustrates our measures for these metrics.

Notably, the results vary less than 1% among React with Rails, Angular with Django, and KVision with Spring Boot regarding `TE`, `ME`, and `MD`. These numbers might indicate that execution time does not change significantly when executing the same functionalities across different technology stacks. However, to better understand these results, we also investigate execution time for each *Simple* and *Composed* testing scenario.

Table 9 presents the Total Execution Time in Seconds (`TE`) for *Simple* testing scenarios. In this context, Angular with Django records the shortest execution time in 6 out of the 14 scenarios. The **Diff%** column also shows that this stack achieves differences greater than 3%. For instance, regarding `createArticle`, Angular with Django completed this testing scenario in 169.44 seconds, while React with Rails took 175 seconds (3.28% slower), and KVision with Spring Boot, 181.14 seconds (6.46% slower).

This performance advantage is also evident in operations such as `createComment` and `deleteArticle`. Angular with Django takes 99.6 and 91.74 seconds, respectively, to complete, while React with Rails takes 108.03 and 95.78 seconds to end. As discussed in Section 3.1, this time efficiency may be attributed to architectural optimizations, such as Angular's Ahead-of-Time (AOT) compilation [3] and Django's ORM [20].

Additionally, React with Rails outperforms the other stacks in certain testing scenarios. For instance, in `filterTag`, it is 6.60% faster than KVision with Spring Boot and 4.14% faster than Angular with Django. In `loadPagesAndWait`, it shows even greater efficiency—11.81% faster than Angular with Django and 2.18% faster than KVision with Spring Boot. Conversely, KVision with Spring Boot exhibits slower execution in most of the *Simple* testing scenarios. This stack generally requires more time to complete operations, as observed in `login` (101.01 seconds) and `logOut` (100.03 seconds).

For the *Composed* testing scenarios, the results in Table 10 show a contrast compared to the *Simple* ones. In this context, KVision with Spring Boot emerges as the fastest stack in 6 out of 18 operations. For instance, in the `createAndUpdateArticle` scenario, it completes the task in 193.01 seconds, while React with Rails takes 221.97 seconds (15% slower), and Angular with Django takes 211.07 seconds (8.56% slower).

Moreover, Angular with Django is faster in scenarios such as `createAndDeleteArticle` (173.85 seconds) and `createAndDeleteComment` (98.12 seconds). However, its

Table 7: Total Energy Consumption in Watts for *Composed* testing scenarios

Testing Scenario	React with Rails	Diff%	Angular with Django	Diff%	KVision with Spring Boot	Diff%
`createAndCommentArticle`	21.72	0.00%	22.96	5.75%	23.22	6.92%
`createAndDeleteArticle`	22.92	1.46%	22.59	0.00%	23.12	2.34%
`createAndDeleteComment`	27.02	0.76%	26.82	0.00%	27.18	1.35%
`createAndUpdateArticle`	21.20	0.00%	21.85	3.07%	22.85	7.79%
`favoriteAndUnfavoriteArticle`	27.30	0.01%	27.30	0.00%	27.52	0.83%
`loginAndCreateArticle`	22.72	1.84%	22.31	0.00%	22.76	1.99%
`loginAndDeleteArticle`	26.68	2.79%	25.95	0.00%	26.78	3.19%
`loginAndDeleteComment`	26.64	3.09%	25.85	0.00%	26.30	1.76%
`loginAndLogout`	25.87	0.00%	26.29	1.65%	26.32	1.73%
`loginAndUpdateProfile`	25.53	1.89%	25.06	0.00%	25.14	0.34%
`loginCreateAndDeleteArticle`	23.22	2.14%	22.73	0.00%	22.98	1.09%
`loginCreateAndDeleteComment`	26.42	0.02%	26.51	0.34%	26.42	0.00%
`loginCreateAndUpdateArticle`	21.15	0.00%	22.57	6.71%	22.85	8.07%
`loginCreateArticleAndComment`	23.08	22.46%	18.85	0.00%	22.87	21.35%
`loginCreateComment`	26.85	3.20%	26.01	0.00%	26.12	0.41%
`LoginFiltertagAndComment`	26.52	5.59%	25.11	0.00%	25.71	2.39%
`singUpAndLogin`	25.40	1.69%	25.49	2.03%	24.98	0.00%
`singUpLoginAndLogout`	24.99	3.37%	24.18	0.00%	24.98	3.33%

Table 8: Time results for each technology stack

Stack	React with Rails	Angular with Django	KVision with Spring Boot
TE	4098.57	4087.90	4069.09
ME	128.08	127.75	127.16
MD	109.80	112.45	110.67

time efficiency decreases in more complex scenarios, like `loginCreateArticleAndComment` (241.25s), where KVision with Spring Boot performs 17.14% faster. In contrast, React with Rails shows higher execution times in *Composed* scenarios, including `createAndCommentArticle` (204.74 seconds) and `loginCreate-ArticleAndComment` (241.25 seconds).

Finally, in response to **RQ2**, and based on the results discussed in this section, there is no definitive best choice for the Real-World App in terms of time efficiency among the three technology stacks, as each demonstrates strengths in different contexts. Angular with Django performs best in most *Simple* scenarios, achieving the fastest execution time in 8 out of 14 cases. In contrast, KVision with Spring Boot stands out in more *Composed* scenarios, leading in 9 out of 18. React with Rails delivers balanced performance across both scenario types, but seldom outperforms the other stacks.

3.3 Carbon efficiency

In this section, we aim to answer our third research question - **RQ3**. As stated in Section 2.1, we measure four metrics: Total CO_2 Emissions in Grams ($\mathtt{TCO_2}$), CO_2 Equivalent Emissions in Grams per Second ($\mathtt{CO_2eq/s}$), and Estimated CO_2 Equivalent Emissions in Grams per Year ($\mathtt{CO_2eq/y}$).

Table 11 demonstrates our results for these metrics. Regarding $\mathtt{TCO_2}$, Angular with Django has the lowest total carbon dioxide emissions at 2.7194 grams, while React with Rails produces 1.2% more CO_2 - 2.7519 grams. Additionally, KVision with Spring Boot has the highest emissions at 2.7717 grams, which is 1.92% higher than Angular.

Moreover, the $\mathtt{CO_2eq/s}$ rates further reinforce that Angular with Django is the most energy-efficient among the evaluated technology stacks, emitting only 0.0217 grams of carbon dioxide equivalent per second during the execution of the testing scenarios. In comparison, React with Rails generates approximately 1% more $\mathtt{CO_2eq/s}$, while KVision with Spring Boot emits around 1.75% more to execute the same set of scenarios.

In terms of projected annual carbon dioxide equivalent emissions, Angular with Django once again reports the lowest value, at 685,636 grams. React with Rails emits approximately 0.81% more, while KVision with Spring Boot reaches 696,588.0 grams—an increase of 1.6%. Despite these differences, the overall results remain relatively close. Therefore, we also calculate the $\mathtt{TCO_2}$ for each *Simple* and *Composed* testing scenario individually.

Table 12 shows the $\mathtt{TCO_2}$ for each *Simple* scenario. Angular with Django demonstrates superior efficiency in 6 scenarios (primarily creation and update operations), while React with Rails leads in 5 scenarios (predominantly login and validation processes), and KVision with Spring Boot excels in only the loadPages scenario. The emission values range from 0.0341 grams to 0.1140 grams of carbon dioxide, with `createArticle` generating the highest carbon footprint across the three stacks. Most scenarios show relatively modest differences between stacks, typically ranging from 1% to 9%, with the maximum difference being 14.98% in `loadPages` between Angular with Django and KVision with Spring Boot.

Overall, the total carbon dioxide emission differences among these technology stacks are relatively small and unlikely to be the sole determining factor in technology selection. While Angular with Django appears to have a slight environmental advantage in more scenarios, none of the stacks demonstrates consistent superiority across the evaluated testing scenarios. Additionally, the similar

Table 9: Total execution time in seconds for *Simple* testing scenarios

Testing Scenario	React with Rails	Diff%	Angular with Django	Diff%	KVision with SpringBoot	Diff%
`createArticle`	175.00	3.28%	169.44	0.00%	181.14	6.46%
`createComment`	108.03	8.46%	99.60	0.00%	106.40	6.39%
`deleteArticle`	95.78	4.40%	91.74	0.00%	98.00	6.39%
`deleteComment`	94.99	2.07%	93.06	0.00%	98.64	5.66%
`filterTag`	91.65	0.00%	95.61	4.14%	98.13	6.60%
`loadPages`	78.85	0.00%	89.41	11.81%	80.61	2.18%
`loadPagesAndWait`	70.52	0.00%	70.57	0.07%	70.84	0.45%
`login`	95.87	0.00%	97.73	1.90%	101.01	5.09%
`loginInvalidEmail`	94.43	0.00%	94.64	0.22%	101.27	6.75%
`loginInvalidPassword`	96.00	0.00%	98.72	2.76%	101.70	5.60%
`logOut`	94.68	0.00%	95.75	1.12%	100.03	5.35%
`signUp`	108.05	3.06%	105.91	1.01%	104.84	0.00%
`updateArticle`	111.54	5.87%	105.36	0.00%	111.29	5.33%
`updateProfile`	112.40	0.00%	112.91	0.45%	114.16	1.54%

Table 10: Total execution time in seconds for *Composed* testing scenarios

Testins Scenario	React with Rails	Diff%	Angular with Django	Diff%	KVision with SpringBoot	Diff%
`createAndCommentArticle`	204.74	12.68%	181.70	0.00%	189.31	4.02%
`createAndDeleteArticle`	177.47	2.08%	173.85	0.00%	175.95	1.19%
`createAndDeleteComment`	104.84	6.85%	98.12	0.00%	107.41	8.65%
`createAndUpdateArticle`	221.97	15.00%	211.07	8.56%	193.01	0.00%
`favoriteAndUnfavoriteArticle`	97.98	3.04%	95.09	0.00%	99.02	3.97%
`loginAndCreateArticle`	187.57	4.19%	184.89	2.63%	180.02	0.00%
`loginAndDeleteArticle`	106.74	0.00%	113.60	6.04%	109.28	2.33%
`loginAndDeleteComment`	107.69	0.00%	111.99	3.84%	110.05	2.14%
`loginAndLogout`	103.26	0.00%	105.42	2.05%	106.70	3.22%
`loginAndUpdateProfile`	122.10	0.00%	126.56	3.52%	123.27	0.95%
`loginCreateAndDeleteArticle`	189.65	0.62%	190.38	1.00%	188.48	0.00%
`loginCreateAndDeleteComment`	126.02	6.06%	123.58	3.85%	118.82	0.00%
`loginCreateAndUpdateArticle`	236.12	16.01%	207.23	1.79%	203.53	0.00%
`loginCreateArticleAndComment`	201.53	0.81%	241.25	17.14%	199.90	0.00%
`loginCreateComment`	119.21	5.31%	113.20	0.00%	116.87	3.15%
`LoginFiltertagAndComment`	117.03	0.00%	123.14	4.96%	123.90	5.55%
`singUpAndLogin`	116.92	0.00%	118.08	0.98%	125.81	7.06%
`singUpLoginAndLogout`	129.94	0.19%	148.31	12.55%	129.70	0.00%

Table 11: Carbon emission results for each technology stack

Stack	React with Rails	Angular with Django	KVision with Spring Boot
TCO_2	2.7519	2.7194	2.7717
CO_2eq/s	0.0219	0.0217	0.02208
CO_2eq/y	691.213	685.636	696.588

emission patterns across stacks indicate that optimizing specific operations within any chosen stack might yield more significant environmental benefits than switching between stacks.

Furthermore, Table 13 illustrates the results for *Composed* testing scenarios. KVision with Spring Boot demonstrates superior efficiency in 4 scenarios, particularly outperforming in scenarios involving article creation and updates (e.g., `loginCreateAndUpdateArticle` and `createAndUpdateArticle`), with emissions as low as 0.1205 grams of carbon dioxide. Angular with Django leads in 7 scenarios, showing particular efficiency in comment-related operations and article interactions, such as `createAndCommentArticle` and `loginCreateComment`, with values down to 0.0709 TCO_2. React with Rails achieves the lowest emissions in 5 scenarios, performing well in logout sequences and authentication flows (e.g., `loginAndLogout` and `signUpLoginAndLogout`). The emission values across the evaluated *Composed* scenarios range from 0.0709 to 0.1364 TCO_2, noticeably higher than most *Simple* scenarios due to their compound nature.

The carbon emission differences between stacks remain relatively modest, with most differences falling between 1% and 7%. The most significant difference observed is 11.08% in the `signUpLoginAndLogout` scenario between React with Rails and Angular with Django. Interestingly, each stack demonstrates advantages in specific operation types: KVision performs better in complex article manipulations, Angular excels in comment operations, and React shows strength in authentication flows. None of

Table 12: Total carbon dioxide emission in grams for *Simple* testing scenarios

Testing Scenario	React with Rails	Diff %	Angular with Django	Diff %	KVision with Spring Boot	Diff %
`createArticle`	0.1096	4.46%	0.1049	0.00%	0.1140	7.94%
`createComment`	0.0798	10.83%	0.0720	0.00%	0.0783	8.05%
`deleteArticle`	0.0719	4.81%	0.0686	0.00%	0.0734	6.54%
`deleteComment`	0.0707	2.46%	0.0690	0.00%	0.0732	5.74%
`filterTag`	0.0666	0.00%	0.0675	1.33%	0.0724	8.01%
`loadPages`	0.0590	3.87%	0.0668	14.98%	0.0568	0.00%
`loadPagesAndWait`	0.0349	2.35%	0.0341	0.00%	0.0353	3.40%
`login`	0.0695	0.00%	0.0723	3.90%	0.0740	6.08%
`loginInvalidEmail`	0.0672	0.00%	0.0681	1.32%	0.0738	8.94%
`loginInvalidPassword`	0.0696	0.00%	0.0706	1.42%	0.0740	5.95%
`logOut`	0.0685	0.00%	0.0691	0.87%	0.0733	6.55%
`signUp`	0.0746	1.36%	0.0736	0.00%	0.0753	2.26%
`updateArticle`	0.0769	2.67%	0.0749	0.00%	0.0782	4.22%
`updateProfile`	0.0756	0.00%	0.0782	3.32%	0.0809	6.55%

the stacks maintains a consistent carbon dioxide emissions advantage across the evaluated *Composed* scenarios. Such results suggest that environmental impact is more dependent on specific operation types than on the technology stack.

Lastly, in response to **RQ3**, we observe that the three evaluated stacks exhibit relatively similar carbon dioxide efficiency for the RealWorld App. While there are differences in the values of `TCO`$_2$, `CO`$_2$`eq/s`, and `CO`$_2$`eq/y`, these variations are minor—approximately 1.6% between the most and least efficient cases. This trend remains consistent when analyzing the *Simple* and *Composed* scenarios separately. Overall, our findings suggest that Angular with Django tends to emit the least carbon dioxide. Specifically, Angular with Django stands out in CRUD operations, whereas React with Rails performs particularly well in authentication scenarios. Finally, despite having the highest total emissions, KVision with Spring Boot demonstrates relative efficiency in handling complex workflows and intensive page loads.

3.4 Statistical differences

To answer **RQ4**, we conduct an analysis of variance (ANOVA)[59]. The results indicate no statistically significant differences among the three evaluated technology stacks, as all p-values across the testing scenarios remain consistently above the 0.05 threshold. Additionally, we apply effect size calculations using Cohen's d[14] for all pairwise comparisons. These results further confirm the absence of substantial differences, with d values ranging from -0.11 to 0.15, indicating only negligible effect sizes.

While Angular with Django tends to exhibit slightly lower energy consumption in several scenarios, and KVision with Spring Boot shows marginally higher usage in others, these differences are not statistically significant. In most cases, the percentage variations remain below 3%. Taken together, these findings suggest that energy consumption alone is unlikely to be a decisive criterion when selecting among web application technology stacks.

We also conduct an ANOVA analysis for time efficiency. It confirms that the execution time differences between React with Rails, Angular with Django, and KVision with Spring Boot are not statistically significant ($p > 0.05$ for both *Simple* and *Composed* scenarios). While KVision with Spring Boot shows the best average performance in *Simple* scenarios (102.5 seconds) compared to React with Rails (102.8 seconds) and Angular with Django (101.5 seconds), the maximum percentage difference between stack means was only 1.3%. For *Composed* scenarios, the differences are similarly small, with mean execution times of 149.1, 150.4, and 146.8 seconds respectively (maximum difference of 2.5%). The effect sizes (Cohen's d) between all pairs of technology stacks fell below 0.2, indicating negligible practical significance. These statistics support the conclusion that performance alone should not be the decisive factor when selecting among these web application technology stacks.

Furthermore, for the ANOVA testing regarding carbon dioxide emissions, we observe that differences are not statistically significant, with p-values of 0.6350 for *Simple* testing scenarios, 0.5102 for *Composed* scenarios, and 0.4381 for all scenarios combined. Therefore, all p-values are larger than 0.05. In summary, we cannot indicate a technology stack superior in terms of carbon dioxide efficiency. Further analysis of effect sizes using Cohen's d confirms the minimal practical significance of these differences. In fact, when all scenarios are combined, they present rates below the threshold of 0.2. For example, React with Rails versus Angular with Django is 0.0345. This statistical analysis suggests that the roughly 1.6% difference in overall emissions between stacks is neither statistically significant nor practically meaningful, indicating that environmental impact should not be the primary decision factor when choosing between these technologies for the RealWorld App.

To explore potential correlations among energy consumption, execution time, and carbon dioxide emissions, we apply the Pearson correlation test [8]. The results reveal strong positive correlations (coefficients > 0.99) across all evaluated metrics for the three technology stacks analyzed: React with Rails, Angular with Django, and KVision with Spring Boot. These high correlation values suggest that increases in execution time are closely associated with increases in energy consumption and carbon emissions. For instance, Angular with Django shows the lowest total energy consumption

Table 13: Total carbon dioxide emission in grams for *Composed* testing scenarios

Testing Scenario	React with Rails	Diff%	Angular with Django	Diff%	KVision with SpringBoot	Diff%
`loginCreateAndUpdateArticle`	0.1364	7.38%	0.1277	0.55%	0.1270	0.00%
`createAndUpdateArticle`	0.1285	6.65%	0.1260	4.38%	0.1205	0.00%
`loginCreateArticleAndComment`	0.1271	2.35%	0.1242	0.00%	0.1249	0.60%
`createAndCommentArticle`	0.1215	6.60%	0.1140	0.00%	0.1201	5.08%
`loginCreateAndDeleteArticle`	0.1202	1.67%	0.1183	0.00%	0.1183	0.03%
`loginAndCreateArticle`	0.1165	4.04%	0.1127	0.71%	0.1119	0.00%
`createAndDeleteArticle`	0.1111	3.56%	0.1073	0.00%	0.1111	3.44%
`loginCreateAndDeleteComment`	0.0910	6.18%	0.0895	4.25%	0.0857	0.00%
`loginCreateComment`	0.0874	8.57%	0.0805	0.00%	0.0834	3.48%
`loginAndUpdateProfile`	0.0852	0.59%	0.0866	2.19%	0.0847	0.00%
`LoginFiltertagAndComment`	0.0848	0.36%	0.0845	0.00%	0.0870	2.87%
`loginAndDeleteComment`	0.0784	0.00%	0.0791	0.88%	0.0790	0.76%
`loginAndDeleteArticle`	0.0778	0.00%	0.0805	3.35%	0.0799	2.63%
`createAndDeleteComment`	0.0774	7.65%	0.0719	0.00%	0.0797	9.79%
`favoriteAndUnfavoriteArticle`	0.0730	2.96%	0.0709	0.00%	0.0745	4.83%
`loginAndLogout`	0.0730	0.00%	0.0758	3.69%	0.0768	4.95%
`signUpAndLogin`	0.0811	0.00%	0.0822	1.34%	0.0858	5.48%
`signUpLoginAndLogout`	0.0871	0.00%	0.0980	11.08%	0.0885	1.54%

(`TC` in Table 5) at 794.4133 watts, which corresponds to the lowest total carbon dioxide emissions at 2.7194 grams ($\texttt{TCO}_2$ in Table 11).

Based on our results, we answer **RQ4** stating that the choice among different web application technology stacks in the context of the RealWorld App tends to have minimal effect on energy consumption, execution time, and carbon dioxide emissions.

3.5 Discussion

Surprisingly, our results indicate that the selection of a technology stack should not be the exclusive determinant when aiming for sustainability in web applications, specifically for the RealWorld App. Rather, targeted optimizations—such as minimizing redundant calls, implementing caching mechanisms, and employing lazy-loading—can lead to substantial environmental improvements. This perspective reorients the conventional strategy from redeveloping systems in newer stacks to enhancing critical functionalities within the existing stack.

In this context, further investigation is necessary to determine what factors are crucial for significantly changing energy consumption, execution time, and carbon dioxide emissions. One hypothesis is that improving specific features (e.g., `authentication`) should bring better results when aiming for software sustainability. Another hypothesis regards whether the slightly enhanced energy efficiency observed in Angular with Django could be attributed to Angular's Ahead-of-Time compilation, which yields more optimized executable code, alongside Django's proficient utilization of ORM for database queries. Conversely, KVision with Spring Boot's comparative advantage in composed scenarios may arise from its integration with the Java Virtual Machine, a characteristic that often optimizes sequences of calls through Just-In-Time compilation. Nevertheless, this very feature might introduce overhead in brief executions, which would account for its worse performance in simpler tasks.

At last, we hope that other researchers could also investigate such hypotheses in order to achieve alternative or complementary conclusions.

3.6 Threats to Validity

In this section, we discuss the potential threats to the validity of our study [64].

External validity. Our findings are based on the evaluation of three specific technology stacks. As such, the results may not be directly generalizable to other combinations of technologies. However, the selected stacks—React with Rails, Angular with Django, and KVision with Spring Boot—are widely adopted in web application development [12, 43, 45, 51, 61, 62]. To mitigate this limitation, we intend to expand our analysis in future work by incorporating additional stacks from the Real World application project.

The second threat relates to the selection of testing scenarios (Tables 3 and 4). Although the scenarios cover common web application functionalities—such as sign-up, login, and logout—we cannot claim that our findings generalize to all types of features. Nevertheless, the selected 32 testing scenarios cover the majority of available functionalities from the Real World project across the three evaluated technology stacks. In this sense, our sample likely captures foundational behaviors of typical web applications.

Furthermore, this study relies on a single case study—the Real World app [65]. As such, our findings are limited by the specific characteristics of how this project is developed, maintained, and shared within the software community. Nevertheless, the Real World app is a well-established project, supported by over 84 contributors, with more than 81,000 stars and 7,000 forks on GitHub. These numbers reinforce its relevance for studies like ours. In fact, there are a number of research studies that use the Real World app for a myriad of purposes [2, 58, 66, 67].

Last but not least, CodeCarbon [11] uses the Intel Running Average Power Limit (RAPL) tool [18, 35] under the hood. This means that our conclusions apply only to web applications running on an Intel Processor. Therefore, we cannot generalize our results for other processors. In addition, conditions such as room temperature might slightly influence our results. However, RAPL is widely used as a measurement tool in academic work [1, 5, 22, 36].

Construct validity. This research uses 10 metrics, as illustrated in Table 1. Therefore, our conclusions are based on the results of such a set of metrics. In this context, we could come across different results in case we use different metrics. For example, the differences in emissions of greenhouse gases other than carbon dioxide, such as methane [24], across the evaluated stacks could be more significant. Nonetheless, to mitigate this issue, we select the `CO₂eq/s` and `CO₂eq/y` metrics to measure greenhouse gas emissions equivalent to carbon dioxide. Thus, we do not evaluate alternative greenhouse gases separately, but we estimate them within those metrics.

Internal validity. To measure energy consumption (in watts), execution time (in seconds), and carbon dioxide emissions (in grams), we rely on the CodeCarbon tool [11]. Ideally, we should employ multiple tools to reduce the risk of bias introduced by limitations or inaccuracies specific to a single tool. However, we argue that CodeCarbon is a widely adopted and validated solution for measuring these metrics [10, 21, 26, 53]. Thus, we consider its use to be a more reliable and less error-prone choice than developing a custom measurement solution from scratch.

4 Related Work

In this section, we review and discuss related work, highlighting key studies in the field and comparing their approaches and findings with those of our research.

First, Hindle et al. [33] introduce GreenMiner, a framework for measuring energy consumption in Android applications using hardware instrumentation (e.g., Arduino [48]). Unlike their approach, which targets mobile apps with physical measurements, we assess server and client-side web frameworks using software-based tools like CodeCarbon. While both studies aim to promote energy-efficient software, our work provides a complementary perspective by analyzing web architectures rather than mobile environments.

Additionally, Georgiou et al. [25] evaluate the energy consumption and runtime performance of 14 programming languages across multiple platforms: embedded systems, laptops, and servers. Their study relies on controlled benchmarks from the Rosetta Code [13], whereas ours uses the RealWorld app [65] to simulate practical web application scenarios. Although both works address energy efficiency, our focus on technology stacks extends their findings to web development.

Pereira et al. [57] analyze energy consumption, execution time, and memory usage across 27 programming languages using benchmarks from the Computer Language Benchmarks Game [28]. While their study focuses on algorithmic tasks at the language level, our work evaluates full-stack web frameworks (e.g., React with Rails, Angular with Django, and KVision with Spring Boot) in application scenarios using the RealWorld App benchmark. This difference in focus allows us to assess the environmental impact of technology combinations rather than individual languages.

The work of de Macedo et al. [19] compares WebAssembly [32] and JavaScript in terms of energy efficiency and execution time. Their study, which includes both micro-benchmarks and real-world applications (e.g., WasmBoy [63]), demonstrates that WebAssembly can reduce energy consumption by up to 30% in compute-intensive tasks. While their work examines low-level execution formats in browsers, we investigate web applications, providing conclusions at a higher architectural level.

Moreover, Rashid et al. [44] investigate the energy consumption of sorting algorithms implemented in ARM Assembly, C, and Java on a Raspberry Pi [54]. Their analysis highlights how programming language abstraction levels and hardware characteristics impact power usage. In contrast, our study focuses on web applications developed using different technology stacks. Furthermore, we present results related to greenhouse gas emissions, contributing to a better understanding of how usage scenarios can affect environmental impact.

Finally, the study by Gordillo et al. [27] investigates differences in energy consumption across algorithms implemented in various programming languages. Their goal is to empirically identify which languages are more suitable from an energy efficiency standpoint. Alternatively, our work does not focus on specific algorithms executed in isolation. Instead, we analyze energy consumption within usage scenarios of a web application implemented using different technology stacks. Additionally, we extend the evaluation by measuring execution time and carbon dioxide emissions.

5 Conclusion

This work investigated how different web application technology stacks differ with respect to energy consumption, execution time, and carbon dioxide emissions. Unlike prior studies that focus on isolated tasks or algorithmic benchmarks, our approach evaluated real usage scenarios, offering a more practical perspective for developers and decision-makers concerned with reducing software environmental impact.

Our results indicated that the choice of web application technology stack, among the three evaluated, has minimal influence on energy consumption, execution time, and CO_2 emissions. Variations across the stacks remain consistently low, and we found no statistically significant differences. This suggests that simply switching technology stacks may not be an effective strategy for reducing environmental impact. Instead, efforts may be better directed at identifying and optimizing specific application functionalities.

As future work, we aim to expand this analysis by incorporating additional technology stacks, exploring different types of applications, and investigating the impact of optimizing specific functionalities. In particular, we plan to leverage Large Language Models (LLMs), such as llama-3.3-70b-versatile and deepseek-r1-distill-llama-70b available on GroqCloud [29], to perform targeted optimizations. An interesting avenue for exploration is whether LLMs can outperform human developers in identifying and implementing energy- and carbon-efficient improvements. At last, long-term monitoring of energy and carbon dioxide metrics in production environments could provide a more comprehensive understanding of how software design decisions translate into environmental outcomes over time.

ARTIFACT AVAILABILITY

We provide all artifacts used in this work in our Online Appendix [30].

REFERENCES

[1] Tanveer Ahmad and Dongdong Zhang. 2021. Using the internet of things in smart energy systems and networks. *Sustainable Cities and Society* 68 (2021), 102783. doi:10.1016/j.scs.2021.102783

[2] Kalev Alpernas, Aurojit Panda, Leonid Ryzhyk, and Mooly Sagiv. 2021. Cloud-scale runtime verification of serverless applications. In *ACM Symposium on Cloud Computing*. 92–107. doi:10.1145/3472883.3486977

[3] Angular. 2025. Ahead-of-time (AOT) compilation. https://angular.dev/tools/cli/aot-compiler. accessed: 2025-03-27.

[4] Angular. 2025. Web framework that empowers developers to build fast, reliable applications. https://angular.dev/. accessed: 2025-03-27.

[5] Lasse F Wolff Anthony, Benjamin Kanding, and Raghavendra Selvan. 2020. Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. *arXiv preprint arXiv:2007.03051* (2020). doi:10.48550/arXiv.2007.03051

[6] Dolores Añón Higón, Roya Gholami, and Farid Shirazi. 2017. ICT and environmental sustainability: A global perspective. *Telematics and Informatics* 34, 4 (2017), 85–95. doi:10.1016/j.tele.2017.01.001

[7] Victor Basili, Gianluigi Caldiera, and Dieter H. Rombach. 1994. The goal question metric approach. In *Encyclopedia of Software Engineering*, John J. Marciniak (Ed.). Wiley, New Jersey, 528–532.

[8] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. *Pearson Correlation Coefficient*. Springer, 1–4. doi:10.1007/978-3-642-00296-0_5

[9] Spring Boot. 2025. Web framework that makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". https://spring.io/projects/spring-boot. accessed: 2025-03-27.

[10] Semen Andreevich Budennyy, Vladimir Dmitrievich Lazarev, Nikita Nikolaevich Zakharenko, Aleksei N Korovin, OA Plosskaya, Denis Valer'evich Dimitrov, VS Akhripkin, IV Pavlov, Ivan Valer'evich Oseledets, Ivan Segundovich Barsola, et al. 2022. Eco2ai: carbon emissions tracking of machine learning models as the first step towards sustainable ai. *Advanced Studies in Artificial Inteligence and Machine Learning* 106, Suppl 1 (2022), 118–128. doi:10.1134/S1064562422060230

[11] Code Carbon. 2025. Track and reduce CO2 emissions from your computing. https://codecarbon.io/

[12] Songtao Chen, Upendar Rao Thaduri, and Venkata Koteswara Rao Ballamudi. 2019. Front-end development in react: an overview. *Engineering International* 7, 2 (2019), 117–126. doi:10.18034/ei.v7i2.662

[13] Rosetta Code. 2025. Rosetta Code. https://rosettacode.org/wiki/Rosetta_Code. Accessed: 2025-03-20.

[14] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. routledge.

[15] Benoit Courty, Victor Schmidt, Goyal-Kamal, MarionCoutarel, Boris Feld, Jérémy Lecourt, LiamConnell, SabAmine, inimaz, supatomic, Mathilde Léval, Luis Blanche, Alexis Cruveiller, ouminasara, Franklin Zhao, Aditya Joshi, Alexis Bogroff, Amine Saboni, Hugues de Lavoreille, Niko Laskaris, Edoardo Abati, Douglas Blank, Ziyao Wang, Armin Catovic, alencon, Michał Stęchły, Christian Bauer, Lucas-Otavio, JPW, and MinervaBooks. 2024. *mlco2/codecarbon: v2.4.1*. doi:10.5281/zenodo.11171501

[16] Cypress.io. 2025. Code Coverage for Cypress. https://github.com/cypress-io/code-coverage.

[17] Cypress.io. 2025. Testing Frameworks for Javascript. https://www.cypress.io/

[18] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory power estimation and capping. In *ACM/IEEE international symposium on Low power electronics and design*. 189–194. doi:doi.org/10.1145/1840845.1840883

[19] Joao De Macedo, Rui Abreu, Rui Pereira, and Joao Saraiva. 2022. Webassembly versus javascript: Energy and runtime performance. In *2022 International Conference on ICT for Sustainability (ICT4S)*. IEEE, 24–34. doi:10.1109/ICT4S55073.2022.00014

[20] Django. 2025. Django Web Framework. https://www.djangoproject.com/. accessed: 2025-03-27.

[21] Pau Duran, Joel Castaño, Cristina Gómez, and Silverio Martínez-Fernández. 2024. GAISSALabel: A tool for energy labeling of ML models. In *ACM International Conference on the Foundations of Software Engineering*. 622–626. doi:10.48550/arXiv.2401.17150

[22] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *ACM/IEEE 45Th annual international symposium on computer architecture*. 383–396. doi:10.1109/ISCA.2018.00040

[23] Eurostat. 2025. Carbon dioxide equivalent. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Glossary:Carbon_dioxide_equivalent

[24] Charlotte Freitag, Mike Berners-Lee, Kelly Widdicks, Bran Knowles, Gordon S Blair, and Adrian Friday. 2021. The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. *Patterns* 2, 9 (2021). doi:10.1016/j.patter.2021.100340

[25] Stefanos Georgiou, Maria Kechagia, Panos Louridas, and Diomidis Spinellis. 2018. What are your programming language's energy-delay implications?. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 303–313. doi:10.1145/3196398.3196414

[26] Johannes Getzner, Bertrand Charpentier, and Stephan Günnemann. 2023. Accuracy is not the only metric that matters: Estimating the energy consumption of deep learning models. *arXiv preprint arXiv:2304.00897* (2023). doi:10.48550/arXiv.2304.00897

[27] Alberto Gordillo, Coral Calero, Mª Ángeles Moraga, Félix García, João Paulo Fernandes, Rui Abreu, and João Saraiva. 2024. Programming languages ranking based on energy measurements. *Software Quality Journal* 32, 4 (2024), 1539–1580. doi:10.1007/s11219-024-09690-4

[28] Isaac Gouy. 2024. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. Accessed: 2024-10-02.

[29] GroqCloud. 2025. Fast LLM inference, OpenAI-compatible. Simple to integrate, easy to scale. https://console.groq.com/docs/models

[30] Guimarães and Andrade. 2025. Online Appendix. https://github.com/Allysson042/sbcars25.

[31] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S. Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2021. Chasing Carbon: The Elusive Environmental Footprint of Computing. In *International Symposium on High-Performance Computer Architecture*. 854–867. doi:10.1109/HPCA51647.2021.00076

[32] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN conference on programming language design and implementation*. 185–200. doi:10.1145/3062341.3062363

[33] Abram Hindle, Alex Wilson, Kent Rasmussen, E Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. 2014. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th working conference on mining software repositories*. 12–21. doi:10.1145/2597073.2597097

[34] InnovationGraph. 2025. Top 50 Programming Languages Globally. https://innovationgraph.github.com/global-metrics/programming-languages.

[35] Intel. 2024. Running Average Power Limit (RAPL) Energy Reporting. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html. Accessed: 2024-01-27.

[36] Congfeng Jiang, Tiantian Fan, Honghao Gao, Weisong Shi, Liangkai Liu, Christophe Cérin, and Jian Wan. 2020. Energy aware edge computing: A survey. *Computer Communications* 151 (2020), 556–580. doi:10.1016/j.comcom.2020.01.004

[37] Eva Kern, Lorenz M Hilty, Achim Guldner, Yuliyan V Maksimov, Andreas Filler, Jens Gröger, and Stefan Naumann. 2018. Sustainable software products—Towards assessment criteria for resource and energy efficiency. *Future Generation Computer Systems* 86 (2018), 199–210. doi:10.1016/j.future.2018.02.044

[38] Keith Kirkpatrick. 2023. The carbon footprint of artificial intelligence. *Commun. ACM* 66, 8 (2023), 17–19. doi:10.1145/3603746

[39] KVision. 2025. Object oriented web framework for Kotlin/JS. https://kvision.io/. accessed: 2025-03-27.

[40] Loïc Lannelongue, Jason Grealey, and Michael Inouye. 2021. Green algorithms: quantifying the carbon footprint of computation. *Advanced science* 8, 12 (2021). doi:10.1002/advs.202100707

[41] Valérie Masson-Delmotte, Panmao Zhai, Anna Pirani, Sarah L Connors, Clotilde Péan, Sophie Berger, Nada Caud, Y Chen, L Goldfarb, MI Gomis, et al. 2021. Climate change 2021: the physical science basis. *Contribution of working group I to the sixth assessment report of the intergovernmental panel on climate change* 2, 1 (2021). doi:10.1017/9781009157896

[42] Medium. 2025. Medium: Where good ideas find you. https://medium.com/

[43] Antonio Melé. 2024. *Django 5 By Example: Build powerful and reliable Python web applications from scratch*. Packt Publishing Ltd.

[44] Luca Ardito Mohammad Rashid and Marco Torchiano. 2015. Energy Consumption Analysis of Algorithms Implementations. In *International Symposium on Empirical Software Engineering and Measurement*. 1–4. doi:10.1109/ESEM.2015.7321198

[45] Anton Moiseev and Yakov Fain. 2018. *Angular Development with TypeScript*. Simon and Schuster.

[46] Ruby on Rails. 2025. A web-app framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern. https://rubyonrails.org/. accessed: 2025-03-27.

[47] Our World in Data. 2023. Our World in Data. https://ourworldindata.org/ Disponível em: <https://ourworldindata.org/co2-and-greenhouse-gas-emissions>. Acesso em: 01/04/2025.

[48] Jonathan Oxer and Hugh Blemings. 2011. *Practical Arduino: cool projects for open source hardware*. Apress.

[49] Showmick Guha Paul, Arpa Saha, Mohammad Shamsul Arefin, Touhid Bhuiyan, Al Amin Biswas, Ahmed Wasif Reza, Naif M. Alotaibi, Salem A. Alyami, and Mohammad Ali Moni. 2023. A Comprehensive Review of Green Computing: Past, Present, and Future Research. *IEEE Access* 11 (2023), 87445–87494. doi:10.1109/ACCESS.2023.3304332

[50] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2021. Ranking programming languages by energy efficiency. *Science of Computer Programming* 205 (2021). doi:10.1016/j.scico.2021.102609

[51] Bambang Purnomosidi Dwi Putranto, Robertus Saptoto, Ovandry Chandra Jakaria, and Widyastuti Andriyani. 2020. A comparative study of java and kotlin for android mobile application development. In *International Seminar on Research of Information Technology and Intelligent Systems*. 383–388. doi:10.1109/ISRITI51436.2020.9315483

[52] Python. 2025. Programming language that lets you work quickly and integrate systems more effectively. https://www.python.org/. accessed: 2025-03-27.

[53] Saurabhsingh Rajput and Tushar Sharma. 2024. Benchmarking emerging deep learning quantization methods for energy efficiency. In *International Conference on Software Architecture Companion (ICSA-C)*. 238–242. doi:10.1109/ICSA-C63560.2024.00049

[54] Raspberry Pi Foundation. 2025. Raspberry Pi: The official website of Raspberry Pi. https://www.raspberrypi.com/.

[55] React. 2025. The library for web and native user interfaces. https://react.dev/. accessed: 2025-03-27.

[56] L. Ross and A. Christie. 2022. Energy Consumption of ICT. https://post.parliament.uk/research-briefings/post-pn-0677/ Disponível em: <https://post.parliament.uk/research-briefings/post-pn-0677/> Acesso em: 05/04/2025.

[57] Francisco Ribeiro Jácome Cunha João Paulo Fernandes Rui Pereira, Marco Couto and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate?. In *ACM SIGPLAN international conference on software language engineering*. 256–267. doi:10.1145/3136014.3136031

[58] Apitchaka Singjai and Uwe Zdun. 2022. Conformance assessment of Architectural Design Decisions on API endpoint designs derived from domain models. *Journal of Systems and Software* 193 (2022), 111433. doi:doi.org/10.1016/j.jss.2022.111433

[59] Lars St, Svante Wold, et al. 1989. Analysis of variance (ANOVA). *Chemometrics and intelligent laboratory systems* 6, 4 (1989), 259–272. doi:10.1016/0169-7439(89)80095-4

[60] Diomidis Spinellis Stefanos Georgiou, Maria Kechagia. 2017. Analyzing Programming Languages' Energy Consumption: An Empirical Study. In *Brazilian Symposium on Programming Languages*. 1–6. doi:10.1145/3139367.3139418

[61] Dave Thomas, David Copeland, and Sam Ruby. 2020. *Agile Web Development with Rails 6*. The Pragmatic Bookshelf.

[62] Craig Walls. 2015. *Spring Boot in action*. Simon and Schuster.

[63] Wasmboy. 2025. Gameboy Emulator Library written in Web Assembly. https://github.com/torch2424/wasmboy.

[64] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[65] Real World. 2025. Real World Example Apps. https://github.com/gothinkster/realworld

[66] Rahulkrishna Yandrapally, Saurabh Sinha, Rachel Tzoref-Brill, and Ali Mesbah. 2023. Carving ui tests to generate API tests and API specification. In *International Conference on Software Engineering (ICSE)*. 1971–1982. doi:10.48550/arXiv.2305.14692

[67] Man Zhang, Asma Belhadi, and Andrea Arcuri. 2022. JavaScript instrumentation for search-based software testing: A study with RESTful APIs. In *Conference on Software Testing, Verification and Validation*. 105–115. doi:10.1109/ICST53961.2022.00022