How Green Is Your Upgrade? Carbon Emission Trends in Python Web Frameworks

Fernando Karchiloff Gouveia de Amorim
fernando.kga@usp.br
Escola de Artes, Ciências e Humanidades (EACH), Universidade de São Paulo
São Paulo, SP, Brazil

Vinicius Renan de Carvalho
vrcarvalho@usp.br
Escola Politécnica, Universidade de São Paulo
São Paulo, SP, Brazil

Daniel Cordeiro
daniel.cordeiro@usp.br
Escola de Artes, Ciências e Humanidades (EACH), Universidade de São Paulo
São Paulo, SP, Brazil

Gerson Sunyé
gerson.sunye@univ-nantes.fr
LS2N, Nantes Université
Nantes, France

Marcelo Medeiros Eler
marceloeler@usp.br
Escola de Artes, Ciências e Humanidades (EACH), Universidade de São Paulo
São Paulo, SP, Brazil

Abstract

Software evolution is a process driven by the need to address new user and environmental requirements, but it can impact quality aspects such as complexity, maintainability, security, and accessibility. However, the impact of software evolution on software sustainability metrics, such as carbon emissions, remains unexplored. In this paper, we investigate the relationship between software evolution and carbon emissions through an empirical study of Python-based web frameworks. We analyze multiple versions of six popular web frameworks by executing a benchmark set and measuring their associated carbon emissions. Our results reveal trends and patterns in emissions across versions, showing that updates do not uniformly lead to improved environmental performance. In particular, we observe that a single dependency change within some frameworks may cause significant fluctuations in carbon emissions. Our findings provide evidence that tracking the impact of updates is crucial to understanding and mitigating the carbon footprint of software supply chains; in addition, it contributes to the growing body of research on sustainable software engineering by emphasizing the need for greater awareness and consideration of carbon emissions during software evolution.

Keywords

Software Sustainability, Green Software, Green Code, Energy Consumption, Carbon Emissions, Software Evolution, Web Frameworks, Python

1 Introduction

Energy consumption and carbon emissions in software ecosystems are becoming more critical, particularly with the rapid growth of AI-based systems [20]. In this context, sustainability metrics have become an important concern in software development; therefore, some studies have focused on understanding implementation strategies to develop more sustainable software [10, 18]. As with other quality attributes, sustainability measures can evolve over time as the software changes, either through direct modifications to the source code or through the addition, replacement, or updating of third-party software components.

In fact, according to Lehman's laws [21, 22], software systems must change over time to accommodate evolving requirements and environmental settings, which can result in a deterioration of overall quality metrics if changes are introduced in an unstructured manner. Recent studies have investigated how software updates have impacted software quality metrics such as structural complexity [9, 33], security [32], feature availability [27], and accessibility [2, 28]. However, to the best of our knowledge, the influence of software evolution on software sustainability metrics such as carbon emissions remains unexplored.

This gap is particularly concerning, given the growing energy demands of modern applications, where even minor inefficiencies can scale significantly, especially in widely adopted applications or frameworks. Moreover, developers typically lack visibility into the environmental impact of their decisions, and dependency updates may introduce hidden costs beyond the developers' control. As a result, carbon emissions can vary across versions without developer awareness or intent. Understanding how sustainability metrics evolve as software systems are updated is therefore important for identifying critical changes, supporting informed decision-making, and promoting more sustainable software engineering practices.

To address this gap, this paper presents an empirical investigation demonstrating that sustainability metrics, such as carbon emissions, can fluctuate across software updates, even when changes appear minor, such as dependency upgrades. More specifically, we analyzed multiple versions of six popular Python-based web frameworks by executing a standardized benchmark suite and measuring their associated carbon emissions, revealing substantial variations in these metrics across successive releases. Our investigation was guided by the following research questions:

- RQ1 How do energy consumption and carbon emissions evolve across successive versions of Python REST frameworks?
- RQ2 Are there statistically significant trends in energy consumption and carbon emissions over time for each framework?

RQ3 Do framework versions exhibit significant changes in energy consumption and carbon emissions throughout their evolution?

The main contributions of this work are summarized as follows:

- We conducted a long-term, large-scale empirical study of energy use and carbon emissions in Python web frameworks.
- We systematically identified trends and structural shifts in energy, carbon, and performance metrics over time.
- We provide empirical evidence that software evolution does not necessarily lead to improved environmental efficiency, showing that some frameworks exhibit increasing carbon emissions over time.

This paper is organized as follows. Section 2 presents some related work. Section 3 details our experimental setup. Section 4 outlines the results of our experiments, while Section 5 discusses our findings. Finally, Section 6 presents some concluding remarks and future directions.

2 Related Work

This section presents the related work centered on software performance, energy consumption, and carbon emissions.

Bednarz and Miłosz [1] compared Django, Flask, and FastAPI in the context of REST API development, measuring request throughput and response times for CRUD operations under controlled Docker-based environments. Their results confirm that FastAPI outperforms the other frameworks in throughput scenarios, while also showing that ORM usage introduces measurable overhead in endpoints without database interaction.

Di Meglio and Starace [26] evaluated five REST API frameworks in three programming languages: Java, JavaScript/TypeScript, and Python. Using multiple execution environments, including both traditional runtimes such as CPython and OpenJDK, and also alternatives such as GraalVM, Bun, and PyPy. By incorporating energy consumption and CO_2 emissions as primary metrics alongside performance, their findings show that the choice of framework and execution environment has a significant impact on both performance and resource efficiency, and that even modest differences in energy consumption can accumulate many kilowatt-hours per year in production deployments.

Fister Jr. et al. [7] investigated the carbon footprint implications of performance bugs, showing through controlled experiments on C++ programs that inefficient implementations, such as unnecessary memory reallocations, can substantially increase both execution time and carbon emissions. If applied to frameworks, it is possible to expect that performance bugs can accumulate carbon emissions throughout the year.

Guimarães and Andrade [10] investigated the energy consumption, execution time, and carbon dioxide emissions of a benchmark web application, the RealWorld App, implemented in three distinct technology stacks (React with Rails, Angular with Django, and KVision with Spring Boot), using CodeCarbon [6] for measurement and covering 32 usage scenarios. Their findings show no statistically significant differences among the stacks, suggesting that targeted optimizations within a given stack may yield greater environmental benefits than replacing the stack altogether.

These studies show that framework selection and configuration as engineering decisions have consequences. However, by testing each framework or stack using a single fixed version, the gap in how energy, throughput, and carbon emissions evolve across the version history of a framework remains unaddressed.

Hindle [13] proposed the Green Mining methodology, which relates software changes and configuration to power consumption by combining repository mining with empirical energy measurement. Applied to Firefox, Vuze, and rTorrent, this methodology demonstrated that software changes can cause detectable power consumption regressions and that certain object-oriented software metrics correlate with power usage, providing empirical support for the idea that the evolution of a software artifact leaves a measurable trace in its energy profile. Following this finding, Mancebo et al. [23] conducted an empirical study examining the relationship between maintainability metrics and energy consumption in four versions of the Redmine project management tool, following the Green Mining methodology. They found a statistically significant positive correlation between total lines of code and processor energy consumption, suggesting that software growth tends to increase energy demands over time.

However, both studies focus on general-purpose applications evaluated in sparse versions and do not address web frameworks, throughput metrics, or carbon emissions. The present work extends this area of study by applying a longitudinal benchmarking approach to a set of Python web frameworks, covering dozens of versions per framework, and examining energy efficiency, time efficiency, and carbon emissions using a standardized workload that remains identical across all evaluated versions to ensure comparability.

Table 1 summarizes the main characteristics of the reviewed studies. The column *Performance* refers to the measurement of metrics such as latency and throughput. The columns *Energy* and CO_2 denote whether the study explicitly measures energy consumption and carbon emissions, respectively. Finally, the column *Longitudinal* indicates whether the study analyzes multiple versions of the same system over time, rather than a single snapshot.

3 Experimental Setup

This study analyzes the evolution of energy consumption and carbon emissions in Python REST frameworks over time. To achieve this, a controlled and repeatable benchmarking setup was required, in which all frameworks were evaluated under the same workload. In the following, we describe the experimental setup, including the selected frameworks, comprising well-established options (e.g., Aiohttp [16], Django [8], FastAPI [30], and Starlette [3]), as well as emerging ones (BáiZé [31] and Muffin [17]), along with the benchmarks, tools, and metrics we adopted in this investigation.

3.1 Benchmark

To standardize the workload executed by each selected framework, a publicly available benchmark suite[1] was used as the basis for structuring the experiments. In this framework, for several libraries, they implemented the different REST endpoints (i) a simple *html* hello-world page, (ii) the **upload** of a file, and (iii) an **api** call

[1] https://klen.github.io/py-frameworks-bench/

Table 1: Comparison of related work and their main characteristics

Work	Object of Analysis	Performance	Energy	CO_2	Longitudinal
Bednarz & Miłosz [1]	REST APIs	✓			
Di Meglio & Starace [26]	Web frameworks	✓	✓	✓	
Fister Jr. et al. [7]	C++ programs	✓	✓	✓	
Guimarães & Andrade [10]	RealWorld App	✓	✓	✓	
Hindle [13]	Software systems		✓		✓
Mancebo et al. [23]	Redmine		✓		✓
Our Work	Python web frameworks	✓	✓	✓	✓

coded as a post request. In order to achieve library performance, they submitted each one to a stress test performed by the HTTP benchmarking tool WRK.[2] The original goal of this benchmark was to discover which library processes more requests.

In that setting, the number of requests varies depending on each framework's performance. This behavior is suitable for throughput comparisons, but it does not align with the requirements of this study. To address this limitation, WRK was replaced by the `hey` tool,[3] which allows a fixed number of requests to be specified. In addition, CodeCarbon [6] was integrated to measure energy consumption and carbon emissions during execution. CodeCarbon was initialized and triggered at the beginning of each execution run, allowing measurements to be collected independently for each repetition. The CodeCarbon geographical location was configured to match the machine's location, in this case, the United States of America (USA), where the AWS region is located. The measurement process was integrated directly into the application code, ensuring that only the workload's execution phase was captured. Later, the benchmark was adapted to support longitudinal analysis. Instead of comparing different frameworks, each is evaluated across multiple released versions, allowing observation of how performance and energy-related metrics change over time.

3.2 Experimental Procedure

Multiple versions of each selected framework were retrieved from their official PyPI repositories,[4] ranging from the first published release of each framework up to March 2026, the most recent version considered in this study. Pre-release and post-release versions (those tagged with `a`, `b`, `dev`, `rc`, or `post` suffixes as defined by PEP 440 [4]) were excluded, resulting in 1,481 stable releases. The most recent version considered was published in March 2026.

Framework versions that failed to build or execute in the defined environment were excluded. Exclusions covered versions that could not be installed due to incompatibilities between the runtime and dependent packages, versions whose benchmark code was no longer compatible with the framework's API, and versions that produced errors during initialization at validation time.

This filtering process resulted in a set of 573 distinct versions across all selected frameworks: FastAPI (191), Django (113), Starlette (130), Aiohttp (61), Muffin (48), and BáiZé (30). Frameworks with an insufficient number of valid versions (fewer than 10) were also excluded from the final analysis, as they do not provide enough data for longitudinal evaluation. The remaining frameworks will be analyzed in Section 4.

Each version of the framework was executed within an isolated Docker container, using Python 3.12.3 as the base environment. No modifications were made to the application code across versions of each framework, the same implementation was reused to ensure consistency and prevent version-specific bias.

All experiments were conducted on Ubuntu 24.04.4 LTS on an Amazon Web Services (AWS) EC2 instance of type **m7i-flex.large**, which is an Intel Xeon Platinum 8488C with 8 GB RAM and 100 GB gp3 EBS (3,000 IOPS, 125 MB/s throughput).

For each version of the framework, the three workloads (API, HTML, and upload) were executed sequentially within the same container. Each workload was executed with a fixed number of 30,000 requests per run (10,000 per endpoint) and 64 concurrent connections. A total of 20 independent runs were performed for each version. Warm-up effects were not considered in the measurements. The container initialization and termination phases were excluded, and measurements were performed only during workload execution.

Two nonparametric tests suitable for time-ordered data were used to assess the behavior of each framework between versions. The Mann-Kendall [24] with Hamed-Rao correction [12] was applied to identify monotonic trends in metrics collected over time (implemented by the package `pyMannKendall` [14]), and the Pettitt test [29] was used to detect change points, indicating moments where a significant shift in behavior occurs during the evolution of a framework (implemented by the package `pyHomogeneity` [15]). Both tests were applied independently to each framework and metric. In both tests, a confidence level of 95% was adopted.

Moreover, code quality metrics were generated using `radon` [19], considering the change points identified in carbon emissions and the first and final versions of the evaluated frameworks.

The source code for the benchmark and the interactive visualizations can be found in the Artifact Availability section.

3.3 Evaluated metrics

The following list of metrics was considered in our analysis:

- **Energy and Emissions**
  - `emission_energy_consumed_mean`: Total energy consumed by the process during benchmark execution (CPU + RAM),

[2] https://github.com/wg/wrk
[3] https://github.com/rakyll/hey
[4] https://pypi.org

measured in kWh. Represents the aggregate energy cost of the framework.
  - `emission_cpu_energy_mean`: Share of total energy attributed exclusively to the CPU, measured in kWh.
  - `emission_ram_energy_mean`: Share of total energy attributed to RAM, measured in kWh.
  - `emission_emissions_mean`: Estimated equivalent amount of carbon dioxide (CO_2) emissions ($kgCO_2eq$) derived from energy consumption and regional carbon intensity.
- **HTTP Performance `/api` Endpoint (JSON response)**
  - `api_throughput_rps_mean`: Mean throughput in requests per second (RPS) at the JSON endpoint.
  - `api_rt_mean_mean`: Mean response time (s) at the JSON endpoint.
  - `api_rt_p95_mean`: 95th percentile response time (s) on the JSON endpoint.
- **HTTP Performance `/html` Endpoint (HTML response)**
  - `html_throughput_rps_mean`: Mean throughput in RPS on the HTML rendering endpoint.
  - `html_rt_mean_mean`: Mean response time (s) on the HTML endpoint.
  - `html_rt_p95_mean`: 95th percentile response time (s) on the HTML endpoint.
- **HTTP Performance `/upload` Endpoint (file upload)**
  - `upload_throughput_rps_mean`: Mean throughput in RPS at the upload endpoint. Reflects the framework's ability to handle `multipart/form-data` requests.
  - `upload_rt_mean_mean`: Mean response time (s) at the upload endpoint.
  - `upload_rt_p95_mean`: 95th percentile response time (s) at the upload endpoint.

4 Results

This section presents the statistical evidence used to answer **RQ1**, **RQ2**, and **RQ3**. Section 4.1 reports the Mann-Kendall trend analysis results, identifying whether energy consumption, carbon emissions, and performance metrics improve, deteriorate, or remain stable throughout the version history. Section 4.2 presents the Pettitt change-point analysis, locating the specific versions where the most significant structural shifts occurred. Section 4.3 examines each framework individually, summarizing its overall trajectory in carbon emissions. The synthesis of these findings and the direct responses to **RQ1**, **RQ2**, and **RQ3** is provided in Section 5.

Throughout this section, we use *improve*, *deteriorate*, and *remain stable* when interpreting the overall trend of a metric, and reserve directional terms such as *increase* and *decrease* for describing the statistical output of the Mann-Kendall and Pettitt tests. We use *remain stable* when the Mann-Kendall test did not detect a statistically significant monotonic trend ($p \geq 0.05$).

4.1 Mann-Kendall Trend Analysis

A Mann-Kendall Trend Analysis was performed on each metric to identify possible trends. The metrics were divided into three groups: latency (Table 2), energy and carbon metrics (Table 3), and throughput (Table 4). As the latency, energy and carbon metrics are unwanted quantities, a decreasing trend corresponds to improvement and an increasing trend corresponds to deterioration. For throughput, this polarity is reversed, as an increasing trend is desirable. Each cell shows the p-value and trend (indicated by an arrow). If the p-value is above the threshold, there is no trend.

Overall, three frameworks show a predominant deterioration, one shows improvements, one has mixed results, and one remains stable. It is possible to observe that Aiohttp has improved energy and carbon emissions, latency, and throughput on the API and HTML endpoints.

BáiZé remains stable across most metrics, except for Upload throughput, which shows deterioration. Django and FastAPI both exhibit a pattern of deterioration across all metrics. Muffin follows the same pattern, except that API endpoint energy and CPU energy show no identifiable trend. Starlette exhibits a improvement across all energy and carbon emission metrics, as well as in upload latency. However, there is deterioration in API Mean and throughput, also at HTML 95th percentile. Other metrics, HTML throughput, HTML Mean and API 95th percentile, remained stable.

Table 2: Mann-Kendall Trend Analysis — Latency Metrics. In this table, ↑ = increasing; ↓ = decreasing; no symbol = no significant trend. Bold = significant ($\alpha = 0.05$).

Framework	*API Mean (s)*	*HTML Mean (s)*	*Upload Mean (s)*	*API p95 (s)*	*HTML p95 (s)*	*Upload p95 (s)*
aiohttp	↓ **0.0014**	↓ **< 0.0001**	0.5530	↓ **0.0037**	↓ **< 0.0001**	0.4538
baize	0.3973	0.7486	0.1898	0.3582	0.2023	0.2089
django	↑ **< 0.0001**	↑ **< 0.0001**	↑ **0.0005**	↑ **< 0.0001**	↑ **< 0.0001**	↑ **< 0.0001**
fastapi	↑ **< 0.0001**	↑ **< 0.0001**	↑ **0.0127**	↑ **< 0.0001**	↑ **< 0.0001**	↑ **0.0224**
muffin	0.5050	↑ **< 0.0001**	↑ **< 0.0001**	0.2479	↑ **0.0010**	↑ **0.0019**
starlette	↑ **0.0065**	0.1271	↓ **0.0281**	0.1549	↑ **0.0279**	↓ **0.0145**

Table 3: Mann-Kendall Trend Analysis Energy & Carbon Metrics. Where ↑ = increasing; ↓ = decreasing; no symbol = no significant trend. Bold = significant ($\alpha = 0.05$).

Framework	*Total (kWh)*	*CPU (kWh)*	*RAM (kWh)*	*CO_2eq (kg)*
aiohttp	↓ **0.0018**	↓ **0.0045**	↓ **0.0011**	↓ **0.0018**
baize	0.2520	0.3524	0.3023	0.2520
django	↑ **0.0002**	↑ **0.0011**	↑ **0.0001**	↑ **0.0002**
fastapi	↑ **0.0157**	↑ **0.0268**	↑ **0.0087**	↑ **0.0157**
muffin	↑ **0.0224**	0.3580	↑ **0.0322**	↑ **0.0224**
starlette	↓ **0.0062**	↓ **0.0030**	↓ **0.0038**	↓ **0.0062**

4.2 Pettitt Analysis

Given the trends identified by the Mann-Kendall analysis, the Pettitt test was applied to identify the most significant changes across versions. A change point does not imply a trend, it marks a point at

Table 4: Mann-Kendall Trend Analysis Throughput Metrics. Where ↑ = increasing; ↓ = decreasing; no symbol = no significant trend. Bold = significant ($\alpha = 0.05$).

Framework	API (req/s)	HTML (req/s)	Upload (req/s)
aiohttp	↑ **0.0021**	↑ < **0.0001**	0.9575
baize	0.8028	0.3702	↓ **0.0489**
django	↓ < **0.0001**	↓ < **0.0001**	↓ **0.0006**
fastapi	↓ < **0.0001**	↓ < **0.0001**	↓ **0.0193**
muffin	0.7829	↓ **0.0001**	↓ **0.0001**
starlette	↓ **0.0081**	0.1099	↑ **0.0375**

which the metric shifted to a new level and remained there. Table 5 presents the detected change-point version and the relative change (Δ%) for each framework-metric pair.

Two frameworks stand out for the concentration and magnitude of their change points. Django's shift at version 5.0 is the most prevalent: 10 of 13 metrics share this change point, with carbon emissions rising +18.1%, API latency increasing +31.3%, and throughput dropping −28.0%. FastAPI shows a similar concentration at version 0.109.0, affecting all energy metrics (+5.9%) and API latency (+33.5%), with a corresponding throughput drop of −25.8%. In both cases, the change is consistent across energy and performance dimensions simultaneously.

Aiohttp presents a different pattern: improvements are distributed across multiple patch versions of the 3.11 series rather than concentrated in a single release. Version 3.11.14 marks a reduction in carbon emissions (−6.8%) and API latency (−34.6%), while 3.11.5 improved the HTML endpoint. The Upload endpoint, however, deteriorated at version 3.9.4.

Starlette's most significant change point is version 0.13.5, where all energy and carbon metrics dropped by approximately −11%, and upload latency decreased by over −32%. Performance on the API endpoint began to deteriorate from version 0.27.0, with mean response time and throughput worsening, followed by the 95th percentile at version 0.29.0.

BáiZé and Muffin show smaller relative changes across all metrics, with no single dominant change point. In BáiZé, version 0.18.0 concentrates most changes, though the magnitude remains modest (under 10% across all metrics). Muffin's change points are scattered across several versions in the 0.96–1.1 range, with changes consistently below 6%.

4.3 Frameworks

Aiohttp (61 versions, 3.7.0–3.13.3) shows a consistent improvement in energy and carbon emissions, as seen in Figure 1. The Mann-Kendall test indicates a decreasing trend ($p = 0.0018$, $\tau = -0.538$), although the first (3.7.0, 14.71 μkgCO_2eq) and last version (3.13.3, 14.83 μkgCO_2eq) alone do not reflect this, as the reduction occurred mid-series. The Pettitt test identifies version 3.11.14 as the point of reduction ($p < 0.0001$), where the mean emissions dropped from 15.49 μkgCO_2eq to 14.43 μkgCO_2eq (−6.8%), after 37 versions in the first segment and 24 in the second.

BáiZé (30 versions, 0.12.0–0.23.1) remained stable throughout its evolution, as seen in Figure 2. No trend was detected by the Mann-Kendall test ($p = 0.2520$, $\tau = 0.204$). The difference in emissions between the first version (0.12.0, 13.87 μkgCO_2eq) and the last (0.23.1, 13.64 μkgCO_2eq) is negligible. Nevertheless, the Pettitt test identified a localized change point in version 0.18.0 ($p = 0.0096$), where mean emissions increased from 13.99 μkgCO_2eq to 14.53 μkgCO_2eq (+3.8%), after 10 versions in the first segment and 20 in the second. This shift did not sustain enough to establish a new pattern.

Django (113 versions, 3.2–6.0.2) exhibits the strongest and most significant deterioration in carbon efficiency among all frameworks, as seen in Figure 3. The Mann-Kendall test confirms an increasing trend ($p = 0.0002$, $\tau = 0.577$). Emissions rose from 26.33 μkgCO_2eq in version 3.2 to 38.01 μkgCO_2eq in version 6.0.2, a +44% increase. The Pettitt test locates the structural break at version 5.0 ($p < 0.0001$), where the segment mean increased from 32.18 μkgCO_2eq to 37.99 μkgCO_2eq (+18.1%), after 68 versions in the first segment and 45 in the second. A notable divergence between versions 3.2.25 and 4.0 is further examined in Section 5.

FastAPI (191 versions, 0.51.0–0.135.1) also shows a statistically increasing trend in carbon emissions ($p = 0.0157$, $\tau = 0.324$), though less pronounced than Django's, as seen in Figure 4. The first (0.51.0, 14.95 μkgCO_2eq) and last version (0.135.1, 14.52 μkgCO_2eq) suggest stability, masking an internal structural shift. The Pettitt test identifies version 0.109.0 as the change point ($p < 0.0001$), where mean emissions increased from 13.59 μkgCO_2eq to 14.39 μkgCO_2eq (+5.9%), after 95 versions in the first segment and 96 in the second. A notable divergence between versions 0.60.0 and 0.60.1 is further examined in Section 5.

Muffin (48 versions, 0.93.2–1.4.2) shows a slow but statistically significant increase in carbon emissions ($p = 0.0223$, $\tau = 0.228$), as seen in Figure 5. The magnitude is modest: emissions rose from 11.12 μkgCO_2eq in version 0.93.2 to 11.46 μkgCO_2eq in version 1.4.2 (+3%), reflecting gradual accumulation rather than an abrupt change. The Pettitt test identifies version 0.98.5 as the consolidation point ($p = 0.0394$), where the segment mean increased from 11.07 μkgCO_2eq to 11.24 μkgCO_2eq (+1.5%). Notably, version 0.98.4 is absent from the official release history, with the series proceeding directly from 0.98.3 to 0.98.5.

Starlette (130 versions, 0.8.5–0.52.1) is the only framework showing a clear and sustained improvement in carbon efficiency, as seen in Figure 6. The Mann-Kendall test confirms a significant decreasing trend ($p = 0.0062$, $\tau = -0.477$), with emissions falling from 14.86 μkgCO_2eq in version 0.8.5 to 13.10 μkgCO_2eq in version 0.52.1, a −11.88% decrease. The Pettitt test locates this change at version 0.13.5 ($p < 0.0001$), where the segment mean dropped from 14.56 μkgCO_2eq to 12.94 μkgCO_2eq (−11.1%), a reduction maintained across the subsequent 83 versions, representing the majority of Starlette's release history.

5 Discussion

Python web frameworks evolve in response to their respective development communities requirements, and no uniform trajectory

Table 5: Pettitt Test change-point version and relative change $\Delta\% = (\mu_2 - \mu_1)/\mu_1$ at the detected break.

	API			Energy				HTML			Upload		
Framework	rt_mean	rt_p95	rps	cpu	emissions	energy	ram	rt_mean	rt_p95	rps	rt_mean	rt_p95	rps
aiohttp	3.11.14 -34.6%	3.11.14 -26.4%	3.11.11 +47.7%	3.11.14 -11.9%	3.11.14 -6.8%	3.11.14 -6.8%	3.11.14 -5.7%	3.11.5 -13.6%	3.11.5 -14.6%	3.11.5 +15.4%	3.9.4 +10.5%	3.9.4 +11.7%	3.9.4 -9.2%
baize	0.18.0 +5.7%	0.18.0 +7.6%	0.16.0 -2.8%	0.18.0 +7.2%	0.18.0 +3.8%	0.18.0 +3.8%	0.18.0 +3.1%	0.22.1 -10.4%	0.18.0 +9.3%	0.15.0 -3.6%	0.18.1 +7.8%	0.18.1 +10.6%	0.18.0 -3.6%
django	5.0 +31.3%	5.0 +51.3%	5.0 -28.0%	4.2.25 +15.5%	5.0 +18.1%	5.0 +18.1%	5.0 +18.7%	5.0 +31.6%	5.0 +46.1%	5.0 -28.5%	4.2.26 +16.4%	5.0 +25.5%	5.0 -14.7%
fastapi	0.109.0 +33.5%	0.109.0 +24.2%	0.109.0 -25.8%	0.109.0 +8.2%	0.109.0 +5.9%	0.109.0 +5.9%	0.109.0 +5.5%	0.104.1 +19.0%	0.109.0 +10.9%	0.104.1 -16.8%	0.109.0 +0.5%	0.109.0 -3.9%	0.109.0 -3.4%
muffin	1.1.4 +3.5%	0.102.3 +2.7%	0.96.1 +2.7%	0.102.3 +3.8%	0.98.5 +1.5%	0.98.5 +1.5%	0.98.2 +1.6%	0.99.2 +5.7%	0.99.3 +2.9%	0.102.2 -4.7%	0.99.3 +2.7%	0.99.1 +3.3%	0.99.3 -2.6%
starlette	0.27.0 +8.6%	0.29.0 +4.1%	0.27.0 -8.6%	0.13.5 -15.4%	0.13.5 -11.1%	0.13.5 -11.1%	0.13.5 -10.3%	0.13.6 +6.7%	0.27.0 +4.7%	0.16.0 -5.3%	0.13.5 -32.5%	0.13.5 -36.1%	0.13.5 +46.3%

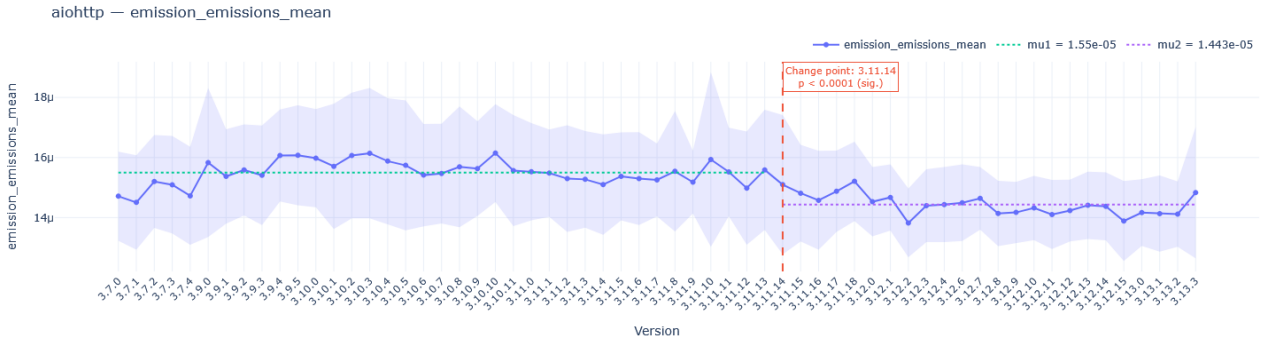

Figure 1: Aiohttp total kgCO_2eq emissions mean by version with change point

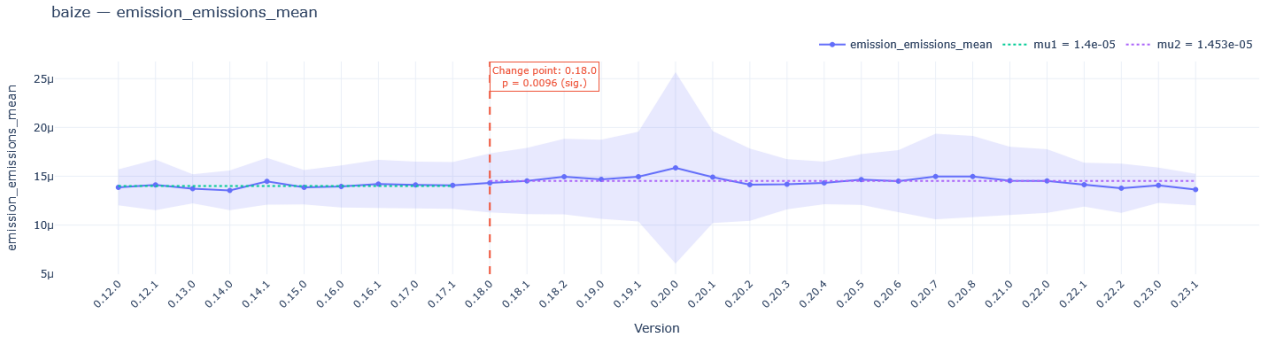

Figure 2: BáiZé total kgCO_2eq emissions mean by version with change point

should be expected across independent projects. What the results reveal, however, is the degree to which these trajectories diverge in energetic terms: some frameworks show sustained deterioration, others consistent improvement, and a subset presents statistically significant change points without a sustained directional trend. This spectrum of outcomes, quantified through Mann-Kendall trend analysis and Pettitt change-point detection, is examined below through the lens of three research questions.

5.1 RQ1: Energy and Emissions Evolution Across Framework Versions

The energy and carbon emissions profiles of the evaluated frameworks fluctuate across their version histories. The direction and magnitude of change vary considerably across frameworks, and no single trajectory characterizes the Python web framework ecosystem as a whole.

Starlette presents the most consistent case of improvement in carbon emissions. Its mean carbon emissions dropped −11.1% at

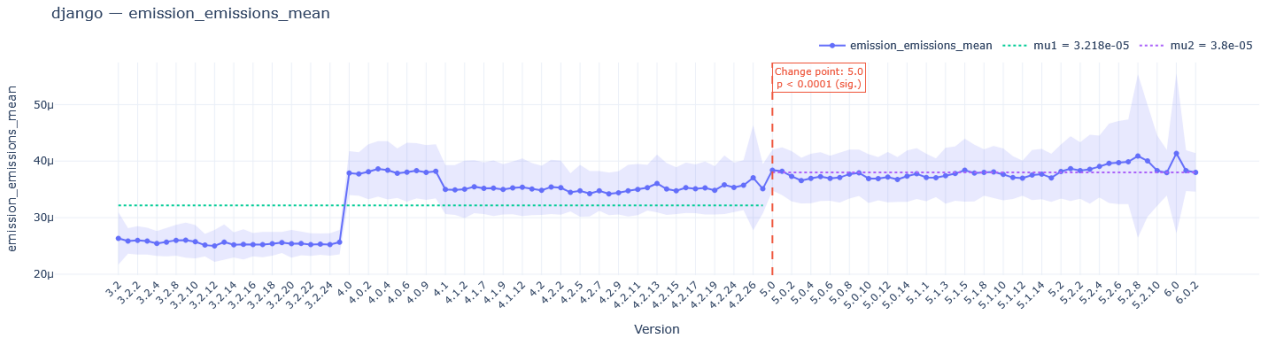

Figure 3: Django total kgCO_2eq emissions mean by version with change point

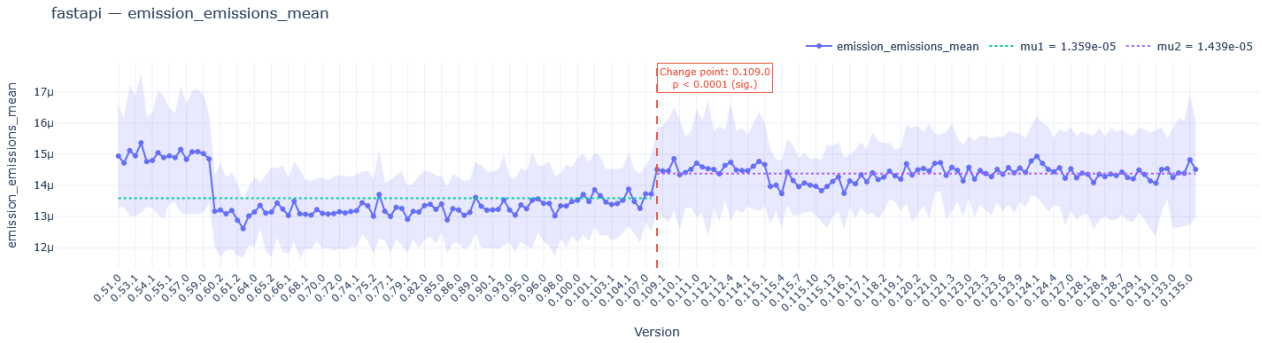

Figure 4: FastAPI total kgCO_2eq emissions mean by version with change point

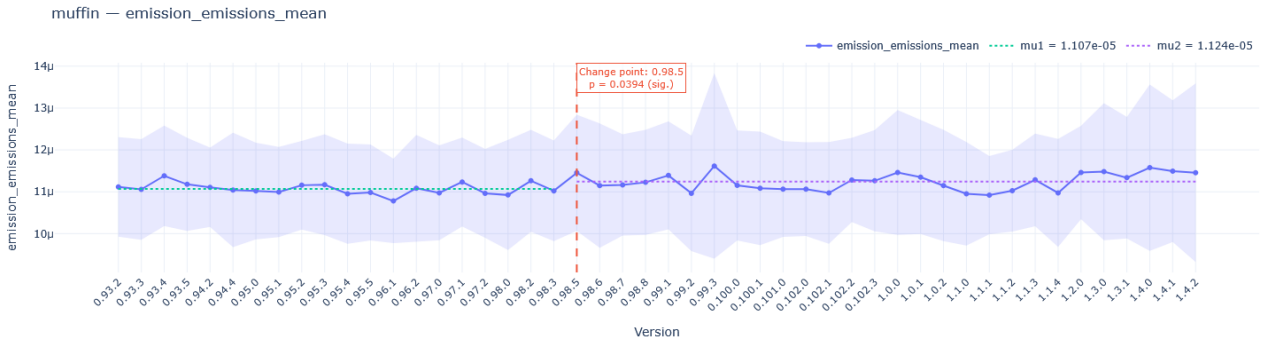

Figure 5: Muffin total kgCO_2eq emissions mean by version with change point

version 0.13.5, and this reduction was maintained across the subsequent 83 versions, representing the majority of Starlette's release history. This makes it the only framework in the evaluated set where efficiency gains are both sustained and consistent across endpoints, demonstrating that energy efficiency can be achievable through framework evolution.

Aiohttp also presents an overall improvement in energy profile, with mean emissions that dropped −6.8% at version 3.11.14. However, this improvement is not uniform across endpoints. The upload endpoint presents a contrasting pattern of deterioration in latency and throughput, indicating that the aggregate reduction in carbon emissions does not extend to all workload types evaluated in this benchmark. Version 3.9.4 introduced changes related to multipart form data compliance and blocking I/O processing during file uploads, which may be associated with the divergent behavior observed in this endpoint.

In contrast, Django presents the most pronounced case of deterioration. Mean carbon emissions increased +18.1% at version 5.0,

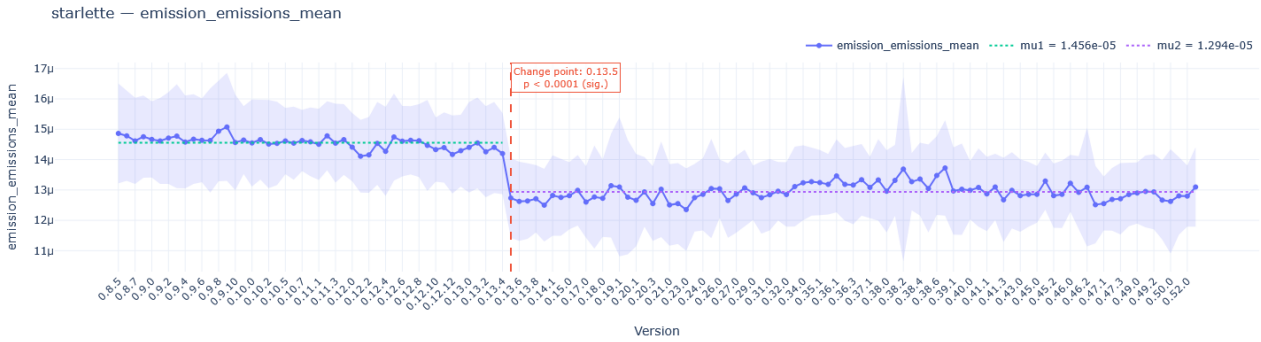

Figure 6: Starlette total $kgCO_2eq$ emissions mean by version with change point

and all throughput metrics show a corresponding decreasing trend across endpoints, indicating that higher energy cost is accompanied by consistent degradation in request processing capacity. A secondary elevation in emissions is observable at the transition from version 3.2.25 to 4.0, coinciding with a major release that introduced numerous breaking changes; however, this shift does not reach statistical significance under the Pettitt test. Understanding its cause would require an in-depth analysis of the Django 4.0 changelog, which is beyond this study's scope.

FastAPI shows a similar deterioration pattern, with mean emissions rising from +5.9% at version 0.109.0, although at a considerably smaller scale than Django. An important characteristic of FastAPI's evolution is its structural dependency on Starlette: FastAPI uses Starlette as its base framework, and inspection of release notes identifies upstream Starlette dependency updates as the sole relevant change at both of FastAPI's detected transitions. The decrease observed between versions 0.60.0 and 0.60.1 coincides with Starlette's own improvement at version 0.13.5, where FastAPI upgraded its Starlette dependency from 0.13.4 to 0.13.6 to address a documented vulnerability in static file handling on Windows[5]. This establishes that FastAPI's energy evolution is not fully autonomous: structural shifts in its carbon emissions profile can be driven by upstream changes in Starlette's development rather than by changes within FastAPI itself.

BáiZé and Muffin represent the last pattern: remain stable, with marginal changes. BáiZé's mean emissions increased +3.8% at version 0.18.0, but this shift did not sustain a new directional pattern, making BáiZé the most energetically stable framework in the evaluated set. Muffin shows the smallest change of all, with mean emissions increasing +1.5% at version 0.98.5, a shift that, while detectable, has negligible practical impact across its tested version range.

Regarding **RQ1**, the evolution of energy consumption and carbon emissions across Python web frameworks does not follow a uniform trajectory, but can be divided into three patterns: **Sustained Improvement**, where Starlette is the most consistent performer in carbon emission reduction, along with Aiohttp, though its gains are not universal across workloads. **Significant Deterioration**, led by Django with a deterioration in carbon efficiency, followed by a drop in request processing capacity. And FastAPI, caused by its energetic profile being influenced by Starlette upstream updates. Finally, **Relative Stability**, where BáiZé and Muffin exhibit only marginal shifts in emissions, but these changes are practically negligible, making them the most stable frameworks evaluated.

[5]https://security.snyk.io/vuln/SNYK-PYTHON-STARLETTE-573266

5.2 RQ2: Statistical Significance of Energy and Emissions Trends

Five of the six evaluated frameworks exhibit a statistically significant monotonic trend in carbon emissions under Mann-Kendall analysis, indicating that the directional changes described in **RQ1** are not attributable to random variation. The exception is BáiZé ($p = 0.2520$, $\tau = 0.204$), for which no significant trend is detected.

Among the frameworks with significant increasing trends, Django presents the strongest signal ($p = 0.0002$, $\tau = 0.577$), followed by FastAPI ($p = 0.0157$, $\tau = 0.324$) and Muffin ($p = 0.0223$, $\tau = 0.228$). The τ values reflect not only statistical significance but also the consistency of the trend: Django's $\tau = 0.577$ indicates that the increasing pattern is present across a large proportion of consecutive version pairs, while Muffin's $\tau = 0.228$ signals a weak and marginally detectable tendency.

Among the frameworks with significant decreasing trends, Aiohttp presents the strongest signal ($p = 0.0018$, $\tau = -0.538$) and Starlette a similarly consistent one ($p = 0.0062$, $\tau = -0.477$). Both values indicate sustained improvement across the majority of consecutive version pairs, lending statistical support to the efficiency gains observed descriptively in **RQ1**.

Regarding **RQ2**, there is a notable detachment between trend significance and change-point detection. While five frameworks show significant trends (Django, FastAPI, and Muffin with increasing trends; Aiohttp and Starlette with decreasing trends), BáiZé shows no trend, and all six frameworks exhibit at least one significant change point. This implies that the two tests provide complementary rather than redundant information. A framework may exhibit a single abrupt change (captured by Pettitt) without a sustained directional movement (captured by Mann-Kendall), or it may exhibit a gradual but consistent trend without a single identifiable inflection point. Therefore, capturing the full picture of a framework's energy evolution requires the combined perspective of both tests.

5.3 RQ3: Significant Change Points in Energy Consumption and Carbon Emissions

All six evaluated frameworks present at least one statistically significant change point detected by the Pettitt test ($\alpha = 0.05$), indicating that none of them maintained a stable behavioral baseline across their full version history, although Muffin and BáiZé show significant change points only in a subset of metrics. Measurable shifts in energy consumption and carbon emissions are therefore a consistent property of framework evolution rather than isolated occurrences.

The change points and their magnitudes have been reported per framework in **RQ1**. To further investigate whether the observed energetic shifts correspond to measurable changes in source code structure, Table 6 presents static code metrics collected at the change point transitions identified by the Pettitt test for each framework. The metrics include source lines of code (SLOC), cyclomatic complexity (CC) [25] aggregated by severity rank, and Halstead difficulty, effort, and bug estimate [11].

Table 6: Source code metrics at Pettitt change-point versions for carbon emissions. Δ = target − base.

Framework (base → target)	Metric	Base	Target	Δ (%)
aiohttp (3.11.13→3.11.14)	SLOC	16,840	16,859	+19 (+0.1%)
	CC total	1,638	1,639	+1 (+0.1%)
	CC high (D/E/F)	14	14	0
	Halstead diff.	5.19	5.20	+0.01 (+0.2%)
	Halstead effort	8670.1	8666.0	−4.1 (−0.0%)
	Bug estimate	15.43	15.43	≈0
baize (0.17.1→0.18.0)	SLOC	2,894	2,928	+34 (+1.2%)
	CC total	365	371	+6 (+1.6%)
	CC high (D/E/F)	0	0	0
	Halstead diff.	3.09	3.11	+0.02 (+0.6%)
	Halstead effort	1430.7	1430.8	+0.1 (+0.0%)
	Bug estimate	2.02	2.02	≈0
django (4.2.27→5.0)	SLOC	105,615	106,635	+1020 (+1.0%)
	CC total	10,352	10,414	+62 (+0.6%)
	CC high (D/E/F)	86	86	0
	Halstead diff.	2.41	2.39	−0.02 (−0.9%)
	Halstead effort	2671.3	2701.1	+29.8 (+1.1%)
	Bug estimate	97.56	98.52	+0.96 (+1.0%)
fastapi (0.108.0→0.109.0)	SLOC	14,751	14,751	0 (0.0%)
	CC total	328	328	0
	CC high (D/E/F)	8	8	0
	Halstead diff.	1.31	1.31	0
	Halstead effort	773.6	773.6	0
	Bug estimate	2.10	2.10	0
muffin (0.98.3→0.98.5)	SLOC	584	582	−2 (−0.3%)
	CC total	43	43	0
	CC high (D/E/F)	0	0	0
	Halstead diff.	1.63	1.63	0
	Halstead effort	324.2	324.2	0
	Bug estimate	0.24	0.24	0
starlette (0.13.4→0.13.5)	SLOC	4,206	4,255	+49 (+1.2%)
	CC total	497	498	+1 (+0.2%)
	CC high (D/E/F)	0	0	0
	Halstead diff.	3.18	3.24	+0.07 (+2.1%)
	Halstead effort	1751.7	1779.4	+27.7 (+1.6%)
	Bug estimate	3.04	3.07	+0.03 (+1.0%)

The source code metrics collected at each change-point transition reveal that structural changes in carbon emissions are not consistently explained by variation in static code complexity. The most striking case is FastAPI, which presents zero variation across all measured metrics between versions 0.108.0 and 0.109.0, yet this transition corresponds to a statistically significant change point in carbon emissions. As established in **RQ1**, the driver is an upstream Starlette dependency update that cannot be detected by static analysis of FastAPI's own codebase. Starlette presents the inverse pattern: a modest increase in code volume (SLOC +1.2%) and Halstead effort (+1.6%) accompanies an −11.1% reduction in mean emissions, suggesting that the added code introduced optimizations whose energetic effect exceeds what complexity metrics alone would predict. Django, the framework with the largest absolute emissions shift at its change point (+18.1%), shows SLOC growth of +1.0% and a corresponding increase in bug estimate, while Muffin and BáiZé exhibit near-zero code variation consistent with their marginal emissions shifts.

Regarding **RQ3**, these observations only partially support the claim by Mancebo et al. [23] that Total Lines of Code is a reliable predictor of energy consumption. Across the full version histories analyzed here, all six frameworks exhibit codebase growth between their earliest and latest benchmarked versions, yet energy trajectories diverge: FastAPI grew from 4,898 to 19,284 LOC (+293.7%) and Django from 130,150 to 160,987 LOC (+23.7%), both consistent with increasing emissions trends, a pattern aligned with Mancebo et al.'s findings. However, Aiohttp expanded from 18,857 to 25,625 LOC (+35.9%) and Starlette from 3,866 to 6,852 LOC (+77.2%), while exhibiting decreasing energy trends, and BáiZé and Muffin present modest growth of +39.9% and +10.3%, respectively, while remaining energetically stable. The evidence indicates that LOC growth is not a reliable predictor of energy consumption: only two of the six frameworks exhibited a corresponding increase in energy consumption, while the remaining frameworks showed stable or decreasing energy consumption despite comparable LOC growth, contradicting prior assumptions. This evidence raises the question of whether other static code metrics could offer better explanatory power, and future work should examine dependency management, architectural decisions, and runtime behavior as potential drivers of energetic trends in framework evolution.

5.4 Threats to Validity

5.4.1 Internal Validity. The primary measurement tool used in this study is CodeCarbon, which estimates energy consumption and carbon emissions based on hardware sensors available through the operating system, including CPU and RAM power draw via Intel RAPL [5]. While CodeCarbon is widely adopted in the literature, it does not capture total system power consumption as an external wattmeter would. A wattmeter measures power at the outlet level, accounting for additional components such as storage, cooling, and power supply inefficiencies that are invisible to software-based sensors. The combined use of CodeCarbon and a wattmeter would provide a more complete picture of the energetic footprint of each framework version, and is left as a direction for future instrumentation.

Additionally, all benchmarks were executed with a fixed server configuration in terms of worker count and threading model. Various configurations may produce different absolute values and potentially alter the relative ordering of frameworks, particularly for frameworks that expose more tunable concurrency parameters.

Also, no warm-up phase was included, the benchmark begins once the container reports a healthy state. Given the scale of requests sent per endpoint and the amount of rounds, it is expected for any cost of initialization to be diluted.

5.4.2 Construct Validity. The benchmark endpoints are purely computational and do not include database access, authentication flows, or other middleware commonly present in production web applications. The evaluated metrics reflect the intrinsic behavior of each framework under synthetic load rather than the behavior of a complete application stack, which can vary depending on its implementation. Results may not generalize to deployments where I/O-bound operations, connection pooling, or ORM layers contribute significantly to energy consumption. Furthermore, the static code metrics reported in Table 6 are limited to complexity and volume indicators. Other dimensions of software quality, such as coupling, cohesion, or architectural patterns, are not captured and may be more directly related to energetic behavior at change points.

5.4.3 External Validity. All experiments were conducted using Python 3.12.3 as the interpreter. No compatibility verification was performed between the tested framework versions and the Python interpreter. The only criterion was whether a given version was successfully built and executed under the defined environment. Identifying the compatible Python version ranges for each framework version remains a direction for future work.

Although different interpreter versions may produce different energy profiles for the same framework version, is expected that the overall trend remains consistent, under the assumption that such variations are more closely tied to the framework internal implementation than to the interpreter itself. Notably, minimum Python version requirements imposed by newer framework releases would restrict the use of older interpreter versions, limiting the breadth of the analysis.

Additionally, this study evaluates CPython 3.12, which does not include just-in-time compilation, experimental JIT support was only introduced in CPython 3.13. Alternative Python implementations, as well as newer CPython versions with JIT enabled, may show substantially different energy characteristics due to differences in compilation strategy and memory management, and are not covered by the present findings.

Finally, the experiments were conducted on a cloud-based instance, which reflects the typical production environment for web frameworks. However, the specific hardware configuration of the cloud instance influences absolute energy consumption values, and results may vary across different instance types or providers. This is a known characteristic of cloud-based energy measurements rather than a limitation specific to this study, and the relative trends observed across framework versions are expected to remain consistent across comparable environments.

6 Conclusion

This study examined how energy consumption and carbon emissions evolve across successive versions of six Python web frameworks, combining Mann-Kendall trend analysis with Pettitt change-point detection to characterize both directional trajectories and structural transitions. The results reveal heterogeneous behavior: Aiohttp and Starlette exhibit statistically significant decreases in energy consumption, Django and FastAPI exhibit increasing trends, while BáiZé and Muffin remain largely stable, with Muffin showing a statistically detectable but practically negligible upward shift.

For practitioners selecting a framework with energy efficiency as a criterion, Aiohttp and Starlette represent the most favorable options among those evaluated. However, framework selection must also account for the functional requirements of the target application, as the observed energy profiles do not reflect the full breadth of features, ecosystem maturity, or suitability for specific workload types that each framework offers.

A notable finding is the dependency relationship between FastAPI and Starlette, where upstream changes in Starlette propagate directly into FastAPI's energy profile. This interaction illustrates that the energy behavior of a framework cannot always be understood in isolation from its dependencies and motivates future work on how inter-framework dependencies mediate energy consumption across versions.

Finally, the analysis of static code metrics at change-point transitions indicates that such measures are insufficient on their own to predict energetic shifts in framework evolution. Future work should investigate additional dimensions, including the influence of Python interpreter versions, alternative server configurations, and workload types that more closely reflect production environments.

Artifact Availability

The source code for the benchmark is available at https://github.com/FernandoKGA/py-frameworks-bench/tree/carbon. The source code for interactive visualizations and plots of the results are available at https://github.com/FernandoKGA/plots-frameworks-bench/tree/cbsoft2026.

Acknowledgments

The authors are members of the Techno-Human Systems of the Future, a pillar of the USP-CNRS International Research Center. This study was financed, in part, by the São Paulo Research Foundation (FAPESP), Brasil. Process Numbers #25/01171-0 and #24/01115-0. We also thank Prof. Fábio Levy Siqueira (Escola Politécnica, Universidade de São Paulo) for his valuable feedback during an early discussion of this work, which helped redirect our research approach toward the direction presented in this paper.

References

[1] Bartłomiej Bednarz and Marek Miłosz. 2025. Benchmarking the performance of Python web frameworks. *Journal of Computer Sciences Institute* 36 (Sep. 2025), 336–341. doi:10.35784/jcsi.7738

[2] Sen Chen, Chunyang Chen, Lingling Fan, Mingming Fan, Xian Zhan, and Yang Liu. 2022. Accessible or Not? An Empirical Investigation of Android App Accessibility. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3954–3968. doi:10.1109/TSE.2021.3108162

[3] Tom Christie. 2026. *Starlette.* https://github.com/Kludex/starlette

[4] Alyssa Coghlan and Donald Stufft. 2013. *PEP 440 – Version Identification and Dependency Specification.* Python Enhancement Proposal 440. Python Software Foundation. https://peps.python.org/pep-0440/ Status: Final.

[5] Intel Corp. 2012. *Intel 64 and IA-32 Architectures Software Developer's Manual.*

[6] Benoit Courty, Victor Schmidt, Sasha Luccioni, Goyal-Kamal, MarionCoutarel, Boris Feld, Jérémy Lecourt, LiamConnell, Amine Saboni, Inimaz, supatomic, Mathilde Léval, Luis Blanche, Alexis Cruveiller, ouminasara, Franklin Zhao, Aditya Joshi, Alexis Bogroff, Hugues de Lavoreille, Niko Laskaris, Edoardo Abati, Douglas Blank, Ziyao Wang, Armin Catovic, Marc Alencon, Michał Stęchły,

Christian Bauer, Lucas Otávio N. de Araújo, JPW, and MinervaBooks. 2024. *mlco2/codecarbon: v2.4.1*. doi:10.5281/zenodo.11171501

[7] Iztok Fister Jr., Dušan Fister, Vili Podgorelec, and Iztok Fister. 2024. Profiling the carbon footprint of performance bugs. In *2024 International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA)*. IEEE, Victoria, Seychelles, 1–7. doi:10.1109/ACDSA59508.2024.10467971

[8] Django Software Foundation. 2026. *Django*. https://github.com/django/django

[9] Jun Gao, Li Li, Tegawendé F. Bissyandé, and Jacques Klein. 2019. On the Evolution of Mobile App Complexity. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, Guangzhou, China, 200–209. doi:10.1109/ICECCS.2019.00029

[10] Allysson Guimarães and Rodrigo Andrade. 2025. Comparing the Environmental Footprint of Web Stacks in the RealWorld Apps. In *Anais do XIX Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 24–35. doi:10.5753/sbcars.2025.14107

[11] Maurice Howard Halstead. 1977. *Elements of Software Science*. Elsevier North-Holland, New York.

[12] K. H. Hamed and A. R. Rao. 1998. A Modified Mann-Kendall Trend Test for Autocorrelated Data. *Journal of Hydrology* 204, 1-4 (1998), 182–196. doi:10.1016/S0022-1694(97)00125-X

[13] Abram Hindle. 2012. Green mining: A methodology of relating software change to power consumption. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, Zurich, Switzerland, 78–87. doi:10.1109/MSR.2012.6224303

[14] Md. Hussain and Ishtiak Mahmud. 2019. pyMannKendall: a python package for non parametric Mann Kendall family of trend tests. *Journal of Open Source Software* 4, 39 (25 7 2019), 1556. doi:10.21105/joss.01556

[15] Md. Manjurul Hussain. 2020. *pyHomogeneity*. doi:10.5281/zenodo.3785287

[16] Nikolay Kim and Andrew Svetlov. 2026. *AIOHTTP*. https://github.com/aio-libs/aiohttp

[17] Kirill Klenov. 2026. *Muffin*. https://github.com/klen/muffin

[18] Christoph König, Daniel J. Lang, and Ina Schaefer. 2025. Sustainable Software Engineering: Concepts, Challenges, and Vision. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 135 (May 2025), 28 pages. doi:10.1145/3709352

[19] Michele Lacchia. 2023. *Radon: Code Metrics in Python*. https://pypi.org/project/radon/ Python package. Available at https://pypi.org/project/radon/.

[20] Chien-Chiang Lee, Jinyang Zou, and Pei-Fen Chen. 2025. The impact of artificial intelligence on the energy consumption of corporations: The role of human capital. *Energy Economics* 143 (2025), 108231. doi:10.1016/j.eneco.2025.108231

[21] M.M. Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076. doi:10.1109/PROC.1980.11805

[22] M. M. Lehman. 1996. Laws of software evolution revisited. In *Software Process Technology*, Carlo Montangero (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 108–124.

[23] Javier Mancebo, Coral Calero, and Félix García. 2021. Does maintainability relate to the energy consumption of software? A case study. *Software Quality Journal* 29, 1 (01 Mar 2021), 101–127. doi:10.1007/s11219-020-09536-9

[24] H. B. Mann. 1945. Nonparametric Tests Against Trend. *Econometrica* 13, 3 (1945), 245–259. doi:10.2307/1907187

[25] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. doi:10.1109/TSE.1976.233837

[26] Sergio Di Meglio and Luigi Libero Lucio Starace. 2024. Evaluating Performance and Resource Consumption of REST Frameworks and Execution Environments: Insights and Guidelines for Developers and Companies. *IEEE Access* 12 (2024), 161649–161669. doi:10.1109/ACCESS.2024.3489892

[27] Maleknaz Nayebi, Konstantin Kuznetsov, Paul Chen, Andreas Zeller, and Guenther Ruhe. 2018. Anatomy of Functionality Deletion: An Exploratory Study on Mobile Apps. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 243–253. doi:10.1145/3196398.3196410

[28] Alberto Dumont Alves Oliveira, Paulo Sergio Henrique dos Santos, Wajdi Aljedaani, and Marcelo Medeiros Eler. 2024. Exploring the Influence of Software Evolution on Mobile App Accessibility: Insights from User Reviews. *Journal of the Brazilian Computer Society* 30, 1 (Nov. 2024), 584–607. doi:10.5753/jbcs.2024.4321

[29] A. N. Pettitt. 1979. A Non-Parametric Approach to the Change-Point Problem. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 2 (1979), 126–135. http://www.jstor.org/stable/2346729

[30] Sebastián Ramírez. 2026. *FastAPI*. https://github.com/fastapi/fastapi

[31] Aber Sheeran. 2025. *BáiZé*. https://github.com/abersheeran/baize

[32] Vincent F. Taylor and Ivan Martinovic. 2017. To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 45–57. doi:10.1145/3052973.3052990

[33] Daniel Thomas. 2025. Sustaining Software Relevance: Enhancing Evolutionary Capacity through Maintainable Architecture and Quality Metrics. In *2025 IEEE/ACIS 23rd International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, Las Vegas, NV, USA, 355–361. doi:10.1109/SERA65747.2025.11154559