UCLouvain

École polytechnique de Louvain

Energy Consumption of Web Servers

A Comparison Between Dynamic and Static Approaches

Author: **Arnaud Jungers**
Supervisor: **Tom Barbette**
Readers: **Clément Delzotti, Robin Dethienne**
Academic year 2024–2025
Master [120] in Computer Science

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Professor Tom Barbette, for his dedication, and invaluable guidance throughout the year. His insightful advice and support have been instrumental in shaping this master thesis.

I am also deeply thankful to Clément Delzotti and Robin Dethienne for kindly accepting to be readers of this master thesis and for their time and valuable feedback.

The experiments presented in this paper were conducted using the Grid'5000 test bed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several universities, as well as other organizations (see https://www.grid5000.fr).

Finally, I extend my appreciation to my family and friends for their unwavering support, encouragement, and patience throughout this journey. Their presence has been a constant source of strength and motivation.

I would aslo like to acknowledge the assistance of AI tools, particularly OpenAI's ChatGPT, which provided support in drafting, refining, and clarifying various sections of this thesis.

Abstract

The growing uses of digital services have led to an increasing concerns regarding the environmental impact, most notably the energy usage, of web hosting. This thesis examines the energy efficiency of different web hosting approaches (static, dynamic, and cached) for different applications using WordPress and Flask-based sites hosted on Apache and Nginx servers, as well as a new approach to build a static version of flask based application on database change. Four test cases were compared, a basic WordPress application and a flask blogging application, one with only get requests, one comparison with the use of post request and one last comparison with authentication. The results are that static implementations always use the least amount of energy and provide the highest throughput. Cached copies offer a good option if static deployment is not possible, but their performance falls when there is increased user activity, i.e., sending POST requests or user authentication. Rebuilding the website upon each database change is a viable approach when fast content updates are required and the frequency of updates is low. However, in scenarios with frequent data modifications, this strategy results in higher energy consumption compared to a dynamic website. Nginx showed to have better power performance compared to Apache and could achieve a higher maximum request per second. These findings illustrate the importance of selecting good host strategies and server settings for energy-saving web applications.

Keywords: energy consumption, web hosting, static websites, dynamic websites, caching, Apache, Nginx, WordPress, Flask

Contents

List of Figures

List of Tables

Glossary

ALD Adaptive Load Distribution. 9

API Application Programming Interface. 13

CDN Content Delivery Network. 1, 8

CMS Content Management System. 3

CPU Central processing unit. 9, 14–16

GHGE global greenhouse gas emissions. 2

HWPC Hardware Performance Counters. 14, 15

ICT Information and Communication Technology. 1, 2, 5

IoT Internet of Things. 3

NAS Network Attached Storage. 8, 44

PWA Progressive Web App. 10

RAPL Running Average Power Limit. 14, 15

rps requests per second. 8, 29, 31, 32, 35, 36, 59

RTT Round Trip Time. 9

SWUT Software Under Test. 6

TDP Thermal Design Power. 15

TLS Transport Layer Security. 9

ultra-low-power board Energy-efficient embedded platforms with low-power MCUs and sleep modes for IoT and battery-powered applications.. 8, 44

Wh watt-hours. 9

Chapter 1

Introduction

In recent years, the increasing demand for digital services has led to a rapid rise in the number of websites and online platforms. This trend has contributed to an exponential rise in web usage globally [17] with a growth of 440% from 2015 to 2021. As the internet continues to expand and integrate further into our daily life, the energy consumption attributed to hosting, maintaining, and transferring web content has become a possible environmental concern. A factor which influence this demand in energy is the design of websites specifically, whether they are implemented as static or dynamic.

In the context of web servers, static websites consist of simple HTML, CSS, and JavaScript files that are sent straight to users without needing database interaction or server-side processing. Usually, Content Delivery Networks (CDNs) are used to cache and distribute these static files, which lowers latency and the server's computational load. A CDN is a geographically distributed network of servers that stores copies of static content closer to end users. By delivering resources from locations near the user, CDNs reduce latency, improve load times, and lower the server's computational load. As a result, static websites typically use fewer resources to function. On the other hand, dynamic websites use web application written in languages like PHP or Python to generate content on the fly. This process often involves querying databases, managing user sessions, rendering content dynamically, and performing other server-side logic, each step contributing to a higher energy consumption. A schematic illustration of how static and dynamic websites handle requests differently is displayed in Figure 1.1.

Web technologies' energy impact is part of a larger environmental dilemma in the Information and Communication Technology (ICT) sector. While ICT innovations have enabled massive efficiencies across several sectors, the industry's own carbon footprint is significant and growing. According to Belkhir and Elmeligi [4], the ICT sector was

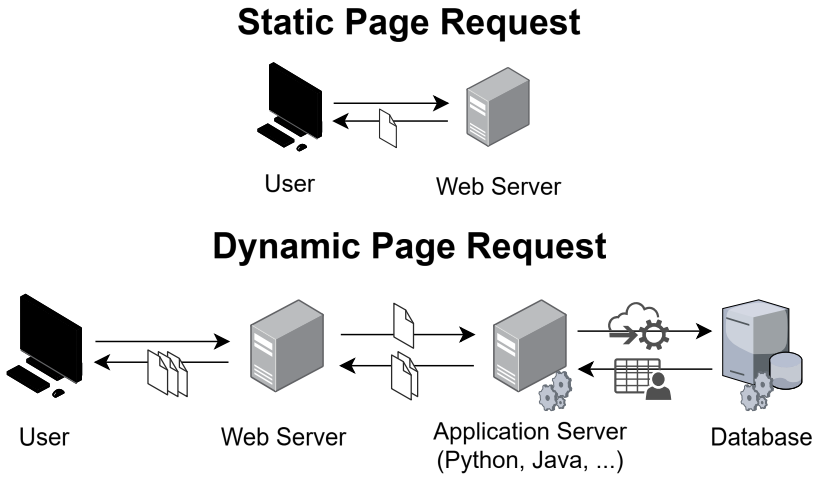


Figure 1.1: A schema which illustrates the key difference between requests sent to a server hosting a static website versus a dynamic website. Static site servers directly return pre-built HTML files. In contrast, dynamic site servers process each request through application logic and databases before generating a custom response.

responsible for approximately 1.6% of global greenhouse gas emissions (GHGE) in 2007. If current trends continue, that proportion may increase to more than 14% of emissions at the 2016 level by 2040. This would put the emissions from the ICT sector on the same level as those from the global transportation sector, stressing how urgent it is to reduce its environmental impact.

A recent study by Istrate et al. [17] highlights the significant environmental footprint of everyday digital activities. The global average consumption of services such as web browsing, social media, video and music streaming, and video conferencing could represent approximately 40% of an individual's carbon budget aligned with the $1.5°C$ climate target. However, the study also highlights a key opportunity: with rapid decarbonization of the electricity supply, the climate impact of digital content consumption could be reduced to just 12% of the per capita carrying capacity by 2030.

Examining data centers, which serve as the physical foundation for all digital services, including websites, a study by Hintemann and Hinterholzer [15] highlights the rising energy requirement brought on by growing computational intensity and data volumes. Their work promotes for the implementation of more energy-efficient hardware, the use of renewable energy sources, and more advanced thermal management solutions such as heat recapture systems.

Further predictions by Koot and Wijnhoven [21] expect data centers to increase their

electricity consumption from 286 TWh in 2016 to as high as 752 TWh by 2030. The estimates are based on varying scenarios, including the stagnation of Moore's law, and the spread of the industrial Internet of Things (IoT). According to the most pessimistic scenario, up to 2.1% of the earth's electricity in 2030 will be consumed by data centers. Their study shows how both technological advancements and the way in which technology is used can greatly affect power consumption.

Within this context, this thesis intends to quantify and compare the energy-consumption of different types of web applications. Through a series of experiments, we will analyze the power consumption of several server configurations when hosting equivalent web applications built using different approaches.

For each web application studied, the following three implementations are compared:

- A fully dynamic version of the application, which serves as a baseline for comparison and involves real-time content generation for each user request.
- A static version of the same application, with all pages pre-rendered and delivered without any runtime computation.
- A dynamic version with server-side caching, designed to reduce the need for repeated computation by storing and reusing previously generated content.

TODO To represent typical use cases, the applications are built using two popular web technologies: WordPress [39], a Content Management System (CMS) built-in PHP, and Flask [10], a lightweight Python web framework. These applications are tested on two common web servers, Apache and Nginx, to study how different servers affect energy usage.

The test cases include a standard WordPress site as well as a series of Flask-based blog applications with increasing functionalities. These range from a simple static blog to a more interactive blog with dynamic content updates via POST requests, and finally, a version featuring user login. These examples are intended to represent a realistic spectrum of modern web applications, excluding more complex microservice-based architectures, which are proposed as a direction for future research.

By exploring these setups, this study aims to provide helpful information on finding a balance between performance, energy consumption and practicality of these approaches. Ultimately, the results can showcase the difference between the energy consumption and max possible rate for each application and help developers make more informed decisions when designing energy conscious web applications.

In this thesis, the following research question are followed:

RQ1: *Compare* the energy consumption of various web hosting approaches (static, dynamic, and cached) and application types (WordPress and Flask with varying features) deployed on servers in the G5k Gros (hardware description shown in 3.1) cluster using Apache and Nginx web servers.

- **RQ1a:** *Compare* the energy consumption of a WordPress site under static, dynamic, and cached hosting approaches on Apache and Nginx.
- **RQ1b:** *Compare* the energy consumption of a basic Flask application under static, dynamic, and cached hosting approaches on Apache and Nginx.
- **RQ1c:** *Compare* the energy consumption of a Flask application with POST requests under static, dynamic, and cached hosting approaches on Apache and Nginx.
- **RQ1d:** *Compare* the energy consumption of a Flask application with user login functionality under static, dynamic, and cached hosting approaches on Apache and Nginx.
- **RQ1e:** *Compare* the energy efficiency of Apache and Nginx web servers with static deployment.

The structure of this thesis is as follows. Chapter 2 analyzes existing literature and current best practices in the field. Chapter 3 outlines the experimental methodology used to evaluate energy consumption across different implementations. The contribution made for this thesis is explained in the chapter Chapter 4. The results and analysis are presented in Chapter 5, followed by a discussion of limitations and potential threats to validity in Chapter 6. The main findings are summarized in Chapter 7, and possible avenues for future work are explored in Chapter 8.

Chapter 2

Related Work

This chapter presents a review of related work relevant to this thesis. First, it discusses the techniques currently employed in the state of the art to estimate and measure energy consumption. Following that, it explores existing approaches aimed at reducing the energy footprint of the IT sector, with a focus on the web domain. Finally, the last chapter show recent studies that compare the performance and energy efficiency of static versus dynamic web approaches.

2.1 Measuring Energy Consumption

To understand the energy use in ICT systems, we need to measure it from different levels of granularity. When we want to know how much energy software uses, special tools are needed. These can be either software or physical devices to ensure we get precise results. Noureddine et al. [25] have put together a thorough guide on the methods and tools available for this task.

Energy measurement techniques can be classified into three main categories: hardware-based measurements, power models, and software-based measurements. Hardware-based are highly accurate, but they lack granularity as they capture the energy of the whole system. Power models employ mathematical or statistical approaches to estimate energy consumption of hardware and software components; however, they tend to be either too generic or platform dependent. In contrast, software-based methods, though less precise, allows to have an energy estimation at the process-level using computational formulas. Examples of such tools include PowerAPI [29], PowerTop [30], and JoulMeter [19], though some software-based methods may still require external hardware power meters. A comparative overview of these methods is presented in Table 2.1.

Name	Energy Measurement	Granularity	Hardware	Category
Powermeter	✓	Hardware	✓	hardware-based
Software Profilers	✗	Software, Methods	✗	software-based
PowerTop	✗	Application	✗	software-based
JoulMeter	✗	Process	✗	software-based
PowerAPI (RAPL)	✗	Process	✗	power-model

Table 2.1: Comparison of energy measurement approaches, highlighting whether they perform direct measurement or estimation, their granularity, hardware dependency, and classification into hardware-based, software-based, or power-model methods.

Beyond these approaches, Preist et al. [31] and Kirkeby et al. [20] extended the analysis by using an interaction design framework to estimate the broader impact of digital services. Their approach considers some indirect energy consumption factors, such as network power usage and resource expenses associated with the use of digital service. This evaluation shows the possibility and necessity of evaluating IT services impact across their entire lifecycle. An example of such a design framework for the YouTube energy estimation is illustrated in Figure 2.1.

Further advancements in this field have been proposed by Ardito et al. [2], who suggest a structured and systematic process for measuring and comparing software energy consumption. Their methodology consists of four phases, each designed to ensure the reproducibility and reliability of energy measurement experiments:

1. **Goal**: Define the context and scope of the Software Under Test (SWUT).
2. **How**: Establish the measurement methodology.
3. **Do**: Perform the measurements.
4. **Analyze**: Evaluate and interpret the collected data.

In addition, the framework explicitly considers potential threats to validity at each stage, contributing to a more systematic and trustworthy approach to measuring energy consumption in software applications.

In the **Goal** phase, the research question is clearly defined, along with the selection of target devices and the execution context for the SWUT. This phase also addresses external validity threats, regarding the generalization of the results to other contexts or systems.

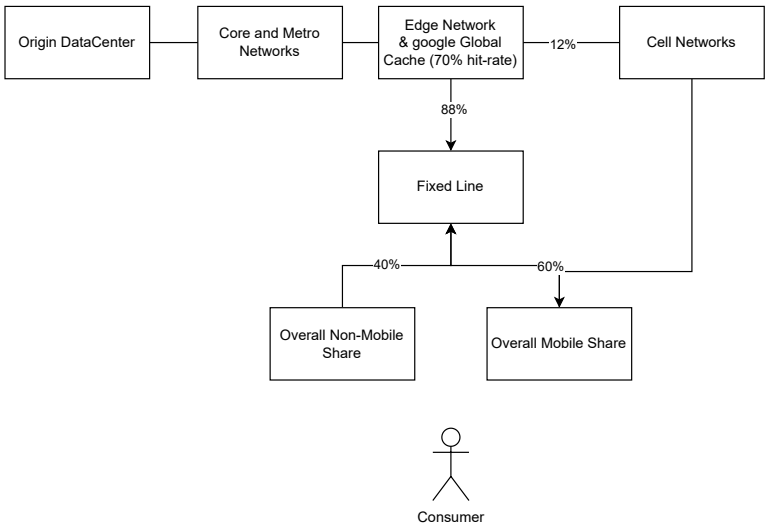


Figure 2.1: An example of a sustainable interaction design framework, proposed by Preist et al., applied to the YouTube web application. The framework uses user network usage data to assess the broader environmental impact of the service.

The **How** phase focuses on detailing the measurement procedure, including the method used to measure/estimate the energy consumption, the measurement frequency, and the format for storing collected data. In this phase, threats such as internal validity (assigning energy consumption correctly to a specific process), construct validity (ensuring accurate measurements), and conclusion validity (ensuring a sufficient number of repetitions) are taken into consideration.

The **Do** phase involves the execution of the experiments, using measurement scripts. A potential issue in this phase is the construct validity, particularly the risk of incorrect implementation of the measurement scripts.

Finally, the **Analyze** phase consists of processing the collected data to address the research questions through the use, statistical test, and graphical representations. At this stage, threats to conclusion validity must be addressed, particularly by selecting appropriate statistical tests to ensure the reliability of the findings.

2.2 Energy Reduction Strategies

This introduces the works that's already done in comparing different approaches to reduce the energy consumption of the IT sector, with a focus on the domain of the web. The first

section is about the approach made on the server side by either using different hardware or the different scheduling algorithms used and the web frameworks used. Finally, a discussion about the approach on the clients side to reduce the energy consumption.

2.2.1 Server-Side Optimization

Using energy-efficient hardware can significantly reduce the environmental impact of web services. Varghese et al. [38] explored the feasibility of using ultra-low-power boards to host both static and dynamic web content. Their study found that a Raspberry Pi could manage about 200 requests per second (rps) when serving static content and around 20 rps for dynamic content. When these boards were used in clusters, they achieved 17 to 23 times more requests per watt compared to traditional systems, showing the significant potential for energy efficiency gains in using clusters of ultra-low-power boards.

Similarly, Everman and Zong [8] evaluated hosting WordPress across three hardware systems: a traditional server, a Network Attached Storage (NAS), and an ultra-low-power board. Their study compared three scenarios: hosting without caching, with caching enabled, and with caching combined with a CDN. Their results indicated that while ultra-low-power boards are well-suited for low-traffic web applications, NAS systems present a more energy-efficient alternative when handling higher traffic volumes. Caching was also found to greatly enhance energy savings in low-power setups. This underscores the importance of using optimization techniques for sustainable web hosting.

Apart from the hardware, the web framework you use on plays a big role in how much energy a website consumes. In a study by De Mander et al. [6] compared Django, Express, Laravel, and Spring Boot, regarding the energy consumption and the response time of the different web frameworks and languages. The findings showed that Express and Spring Boot were the most efficient. These two frameworks used less energy and responded more quickly compared to the others. However, their methodology of measuring total system energy on a laptop rather than energy per process limits the generalizability to dedicated server environments. The previous study is related to a study made by Pereira et al. [27], who investigated energy consumption across different programming languages and demonstrated that the energy consumption of the various languages differs noticeably from one another, and showed that a faster language is not always the most energy efficient. Similarly, Manotas et al. [23] examined the impact of different web servers on energy usage, demonstrating significant variation depending on the web server and the various applications served. Some web server were more energy efficient under specific workload. But this study focused only on Ruby-based web-servers.

Scheduling algorithms also play a crucial role in optimizing energy consumption. Arys and Carlier [3] evaluated two schedulers made using constraint programming with distinct optimization objectives: a Cores-aware scheduler, designed to minimize the total number of active CPU cores, and an Energy-aware scheduler, which minimizes total energy consumption based on real-world a priori energy measurements. Their results indicate that the Energy-aware scheduler outperforms both standard cloud schedulers and the Cores-aware scheduler, achieving energy savings of up to 19% in terms of watt-hours (Wh). Lenhardt et al. [22] demonstrated beforehand that an Adaptive Load Distribution (ALD) strategy reduced energy consumption by 5% to 8% compared to a conventional load balancer. These findings highlight the potential of using energy aware scheduling algorithms to reduce the energy footprint in web hosting environments.

Sapra and Hindle [32] evaluated the energy efficiency of two web servers using HTTP/1.1 and HTTP/2, while also analyzing the impact of Transport Layer Security (TLS) on power consumption. The findings suggest that multiplexing and Round Trip Time (RTT) are key factors in HTTP/2's efficiency. Despite the energy overhead of TLS, HTTP/2 can enhance both performance and energy efficiency in high-latency networks. These results are not peer-reviewed and should therefore be interpreted with caution.

A last possible way to reduce the energy consumption on the server side is explained by Elnozahy et al. [7], who employed two power management mechanisms: dynamic voltage scaling and request batching to lower the energy consumption. The use of both mechanisms can save from 17% to 42% of the Central processing unit (CPU) energy. These results have to be taken with caution, as the study was made in 2003 and CPUs have vastly improved and changed since this period.

You can find a summary of the findings about the energy efficiency on the web on the server side in the Figure 2.2.

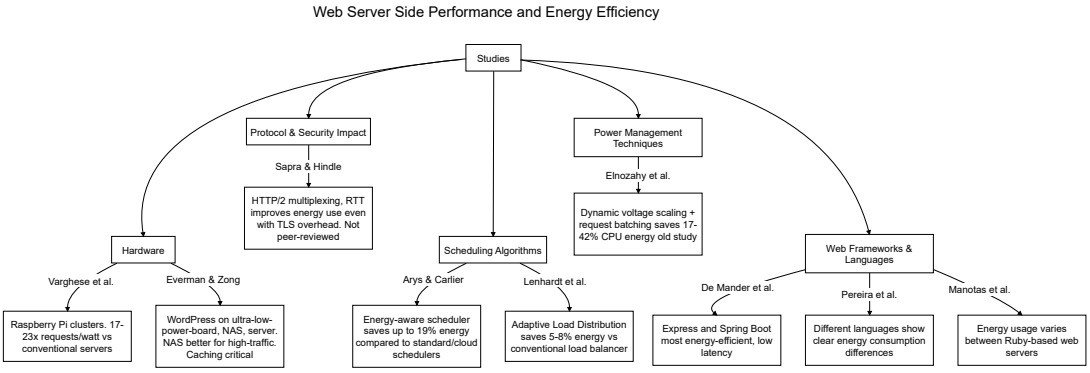


Figure 2.2: Overview diagram summarizing existing research studies which compare energy efficiency from the server-side perspective in web-based application.

2.2.2 Client-Side Optimization

Several studies have looked into what affects energy consumption on the client side and how to reduce it.

Philippot et al. [28] used the Web Energy Archive (WEA) which is a tool that measures how much energy is used when a website loads in a browser. In their study, they found a correlation between RAM usage and energy consumption, which are both directly influenced by the volume of requests sent while loading a website page. They also showed that high Google Page Speed scores do not necessarily translate to lower energy use. For instance, a high-performance website with an elevated Google Page Speed score does not necessarily correlate with low memory or energy consumption. This underscores the need for optimizations that balance both performance and resource efficiency.

Energy consumption also varies significantly depending on the framework used for the user interface. Huber et al. [16] compared mobile implementations with Flutter, Progressive Web App (PWA), Capacitor, React Native, and Native Android, finding that native applications consumed the least energy, while PWAs showed to be a competitive alternative among cross-platform frameworks such as React Native and Flutter. Additionally, they studied the impact of the browser choice and found that the choice of browser significantly affected energy usage, with conflicting results between studies by Huber et al. and van Hasselt et al. [37]. While Huber et al. [16] found that Chrome consumed less energy than Firefox, van Hasselt et al.[37] and Philippot et al. [28] reported the opposite, highlighting inconsistencies in findings regarding browser energy efficiency. These difference may suggest that experimental conditions play a key role in determining results. Similarly, Miettinen and Nurminen [24] found in 2010 that different frameworks could yield up to a 30% variation in energy consumption. However, these results must be interpreted with caution, as the mobile platforms and frameworks have undergone significant evolution since the time of the study, potentially limiting the direct applicability of their conclusions to contemporary contexts. Another study by van Hasselt et al. [37] investigated the energy efficiency of Web Assembly in comparison to JavaScript in a mobile application, demonstrating that Web Assembly generally reduces energy consumption and the runtime of mobile applications.

Beyond software, interface design can influence energy efficiency. Agolli et al. [1] demonstrated that adjusting color schemes by using colors that consume less energy but which aren't different from the base color, particularly for AMOLED displays, led to energy savings of up to 7.8% without significantly affecting user experience. These findings emphasize the potential for client-side optimizations to contribute to overall energy reduction in web applications.

You can find a summary of the findings about the energy efficiency on the web on the

client side in the Figure 2.3.

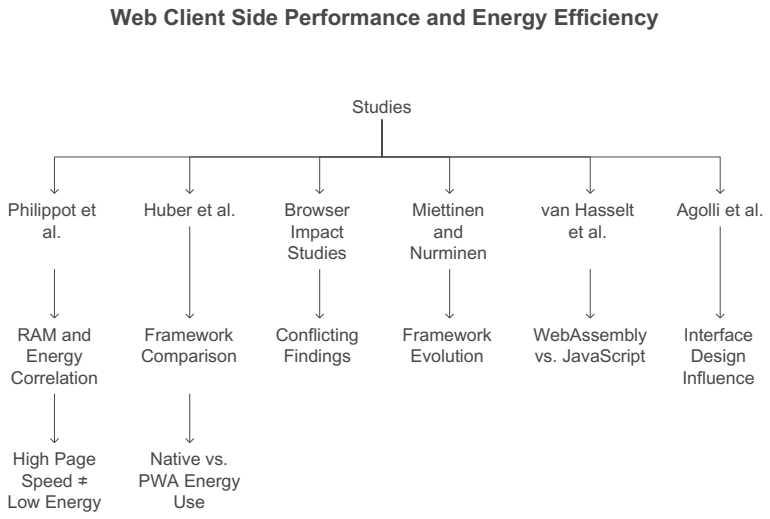


Figure 2.3: Overview diagram summarizing existing research studies which compare energy efficiency from the client-side perspective in web-based application.

2.3 Static vs. Dynamic Performance Comparison

The state of the art about comparing the performance of static and dynamic in web is not well furnished. But one study by Tomiša et al. [36] evaluated the performance of a standard WordPress-based website compared to its static version generated using the WP2Static plugin. Using the Apache Bench tool, they measured metrics such as time per HTTP request and HTTP requests per second on a small virtual machine. Their results showed that the dynamic WordPress site could handle 6.16 HTTP requests per second with an average time of 162.216 ms per request, whereas the static version handled 2218.58 requests per second with an average time of only 0.451 ms per request. These findings demonstrate that the dynamic features of WordPress, such as executing PHP scripts and sending request to the database, are resource-intensive and significantly slower, while converting a WordPress site to a static version improved response time by 97% and increased request handling capacity by a factor of 360. This shows that when using a WordPress application which doesn't rely on the dynamics of WordPress, converting it to a static version gives better performance.

2.4 Gap in current State of The Art

In the state of the art on energy efficiency and comparison, there are some studies, such as [6], that focus on energy consumption across different web frameworks on the backend. Other works [16, 24] also contribute to this area on the frontend side. However, there remains a gap in the literature regarding the comparison of different architectural approaches such as web servers, static versus dynamic content delivery, and various implementation styles like monolithic versus microservices from an energy efficiency perspective. While some studies have studied the performance comparisons of some of these approaches, their energy impact has yet to be thoroughly explored.

Chapter 3

Research Method

This section explains the methodology followed in this thesis to compare the energy consumption of the various implementations evaluated in this thesis, and explain the different tools used in this thesis. It first explains in detail the approach to estimate the energy consumption of the various implementation, the configuration of the server environment, the hardware, and the tools for generating load on each application. Finally, it explains how these individual parts were integrated altogether with npf to collect data across multiple runs. And a last section explaining how the static version of each application is generated.

3.1 Energy Measurement Methods

The estimation of the energy consumption for the various implementations analyzed in this thesis was done using *PowerAPI* [29], an Application Programming Interface (API) to create software-defined power meters which are capable of estimating the power consumption of individual processes within a system.

A software-defined power meter can be seen as a modular application that monitors the power consumption of software processes, either on a single host or across distributed systems. The general architecture of such system, as implemented by PowerAPI, is illustrated in Figure 3.1 and is composed of four modules:

- **Sensor**: Responsible for collecting raw data from hardware events, which is subsequently stored in a database or a CSV file. The sensor must be located on the same host and concurrently with the monitored processes.

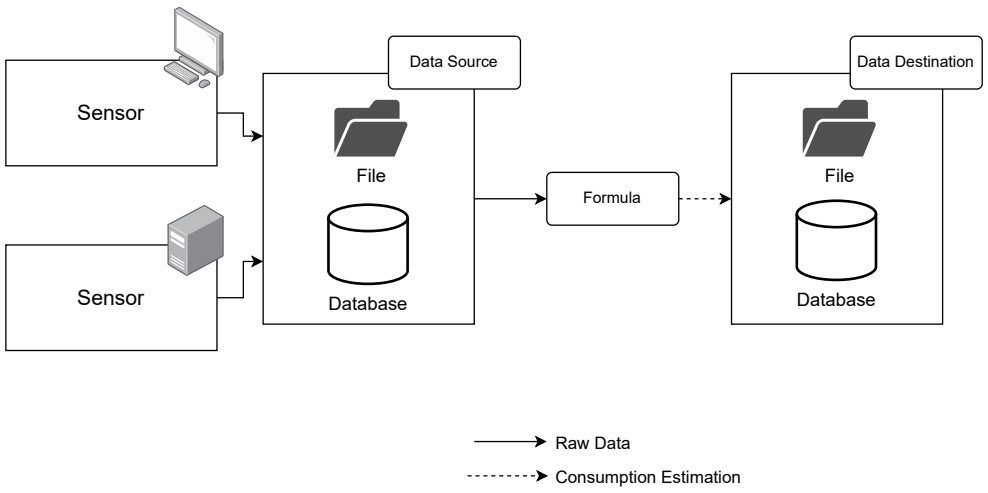


Figure 3.1: General architecture of a software-defined power meter in PowerAPI. Sensors gather raw CPU event data and store it in a database or CSV file. The formula processes this data to estimate per-process energy consumption, with the results exported to a separate storage.

- **Data Source/Destination**: Components responsible for storing the raw data collected by the sensor (source) and for storing the energy estimations produced by the formula (destination).
- **Formula**: This module processes the raw data collected by the sensor to estimate the energy consumption of each running process, exporting the results to the destination storage.

3.1.1 PowerAPI Sensor

The measurement of power consumption begins with the *sensor* module, a software component responsible for collecting raw data from the hardware performance counters of the monitored application. The raw data collection is performed by querying hardware resources on the host machine, and therefore the sensor must be ran on the same host as the monitored processes. The collected data is then stored either in a local or remote database, or in a CSV file, for processing by the formula.

In this study, the Hardware Performance Counters (HWPC) sensor [14], the primary sensor provided by PowerAPI, is used. This sensor collects CPU performance and energy consumption data via the Running Average Power Limit (RAPL) interface, available on most modern processors. It additionally utilizes the `perf` API from the Linux kernel to

gather performance metrics such as TSC, APERF, and MPERF counters. Consequently, PowerAPI is currently only available on Unix-based systems.

The HWPC sensor can be configured to adjust the sampling frequency, which is the rate at which raw event data is collected and averaged over the sampling interval. Users can also specify which system events to monitor, and choosing for all the system event between one monitoring per socket or all-core monitoring per socket, and the container events which are events collected for each running processes.

Per-Process Isolation

To enable per-process power estimation, the sensor uses Linux *control groups* (cgroups), a kernel feature that isolates and allocates system resources to sets of processes. By mapping of monitored processes to specific cgroups, the sensor can collect detailed performance data at the per-process level.

3.1.2 PowerAPI Formula

The power estimation from raw sensor data is performed by the *formula* module. In this study, the *SmartWatts Formula* [34] the default model provided by PowerAPI is used. SmartWatts employs machine learning models to estimate the energy consumption based on the HWPC metrics collected by the sensor.

Accurate estimation requires basic information about the hardware configuration, specifically the CPU base frequency and its Thermal Design Power (TDP), which must be provided in the configuration of the formula.

The SmartWatts Formula [9] builds and maintains separate power models for different hardware components, such as the CPU and DRAM. These models are implemented using Elastic Net regression, and has the possiblity to self-calibrate. When the difference between the estimated energy consumption and the actual energy consumption (measured via the RAPL interface) exceeds a configurable threshold, the model automatically retrains itself using the most recent data samples.

The configuration used for PowerAPI during the experiments conducted in this study are presented in Section A.1.

3.1.3 Comparison Between Hardware Power Meter and PowerAPI Measurements

To validate the reliability of the power estimations produced by PowerAPI, a comparison was done against measurements obtained using an external hardware power meter that monitors the energy consumption of the entire machine. The hardware power meter thus provides a ground truth measurement, which captures all sources of power consumption including CPU, memory, storage devices, cooling systems, etc.

In contrast, PowerAPI focuses exclusively on estimating the energy consumption attributable to the CPU (and optionally DRAM, depending on the configuration).

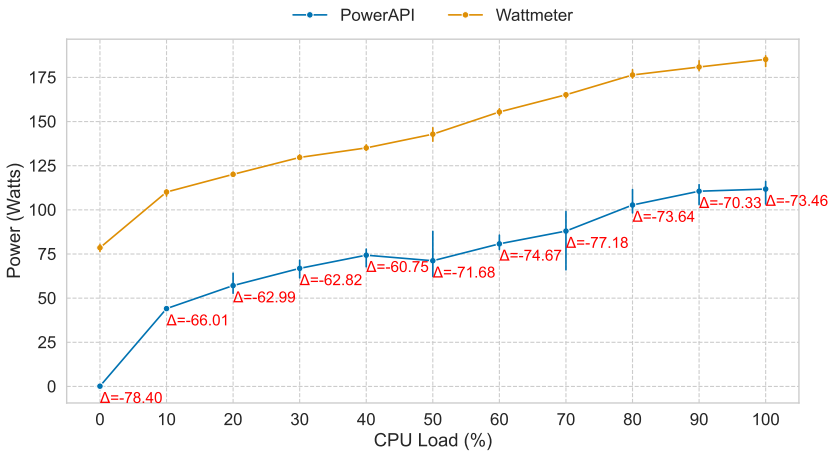
To do this comparison, we ran a script that stressed the CPU for different CPU load using *stress-ng* [35], and compared the results gathered from powerAPI and from the hardware power meter. This experiment was done on the same hardware as the comparison study of the different application and can be found in the Table 3.1.

The results of this experiment are shown in Figure 3.2. Two tests were conducted: one with a single CPU core under stress (Figure 3.2b), to reflect conditions similar to those of the experimenent done in this thesis where the webserver also operates on a single core; and another with all CPU cores stressed (Figure 3.2a). The comparison revealed that PowerAPI consistently reports only a portion of the total system energy consumption, typically missing around 75 watts. This discrepancy is expected, as total system energy includes contributions from components beyond the CPU, such as RAM, storage devices, and cooling systems. We can therefore estimate the total power consumption of the machine with the estimate from powerAPI.

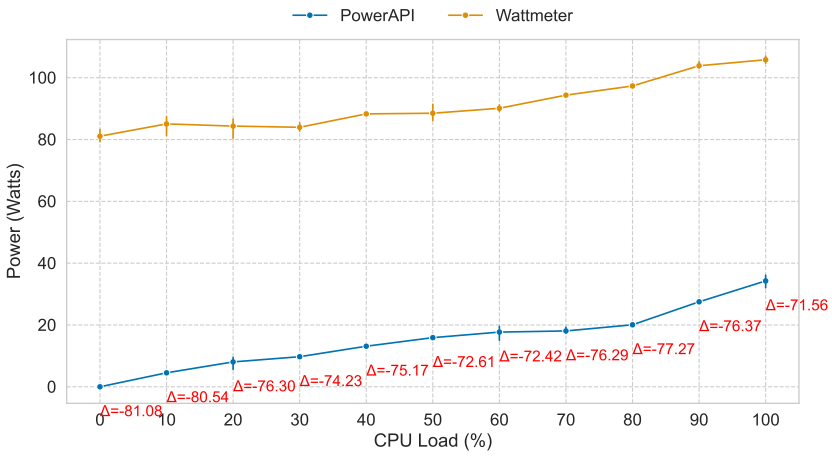
3.2 Web Frameworks & Server Setup

To compare the energy consumption of static and dynamic web approaches, the different experiments were conducted on dedicated servers within the Grid'5000 (G5K) network. Grid'5000 is a large-scale and configurable experimental testbed designed for research in computer science, particularly in parallel and distributed systems, cloud computing, high-performance computing (HPC), big data, and artificial intelligence.

For this thesis two nodes from the Gros cluster of Grid'5000 were used, the hardware configuration of such nodes can be found in the table below Table 3.1. One node was used to generate workload traffic using a workload generator (as described in Section 3.3), while the other ran the target web application alongside the PowerAPI power meter.



(a) Comparison between energy estimations from PowerAPI and the total energy consumption, using every CPU core.



(b) Comparison between energy estimations from PowerAPI and the total energy consumption, using one CPU core.

Figure 3.2: Comparison between energy estimations from PowerAPI and the total energy consumption measured by the hardware power meter. With the delta between the total energy consumption and the CPU energy usage.

Model	Dell PowerEdge R640
CPU	Intel Xeon Gold 5220 (Cascade Lake-SP), x86_64, 2.20GHz, 1 CPU/node, 18 cores/CPU
Memory	96 GiB
Storage	• 480 GB SSD SATA Micron MTFDDAK480TDN
Network	• eth0/eno1: 25 Gbps Ethernet (Mellanox Technologies MT27710 Family, ConnectX-4 Lx), SR-IOV enabled • eth1/eno2: 25 Gbps Ethernet (same as above)

Table 3.1: Hardware characteristics of the Gros cluster node used in the experiment [12]

CPU Core Isolation for Measurement Accuracy

To ensure accurate energy measurements and to minimize measurement interference, CPU core isolation was applied on the server running the web application. Specifically, the Linux kernel was configured to prevent the execution of any system processes on some cores. On the node which has the generation load tolls, the core isolation was also done to have a more consistent load generation.

This configuration ensured that the monitored application processes were isolated from other system activities, such as PowerAPI's sensor and formula modules, and other running processes. As a result, the separation between the application workload and background processes significantly reduced measurement noise, leading to more precise and reliable energy consumption estimations.

While isolating the web-application to a single core limits the maximum number of concurrent requests, in our context it's beneficial as the workload generators used in this thesis do not exceed the request rate that a single-core implementation can handle under our experimental settings.

To ensure the reproducibility of the experiments conducted in this thesis, the information about all the software version used during this thesis is provided in Table 3.2.

For all Flask-based applications used during the thesis, exact package versions are available in the git repository (`https://forge.uclouvain.be/ArnaudJungers/web-server-energy-comparison`) associated with this master thesis.

Software Name	Software Version
WordPress	6.6.2
PHP-FPM	8.3
Flask	3.1.0
Gunicorn	23.0.0
Python	3.12.10
Apache	2.4.58
Nginx	1.24.0
NPF	2.0.3
wrk2	0.2.0
JMeter	5.6.3

Table 3.2: Software versions of the different tools used in this master thesis

3.3 Request Simulation & Load Testing

To evaluate the energy efficiency of the different implementations under some workload, it is essential to simulate web traffic in a controlled and reproducible manner. In this study, two tools were employed to generate HTTP requests at a constant rate and simulate user load: wrk and Apache JMeter. These tools were selected due to their widespread use, flexibility, and complementary capabilities in simulating various types of web server loads.

Wrk [40] is an HTTP benchmarking tool capable of generating a significant number of requests. It is particularly well-suited for low-level performance testing, which offer precise control over concurrency, number of threads, and duration. Wrk requests can be configured with a Lua script. In this work, wrk2 from DeathStarBench was used to simulate high-throughput basic scenarios, focusing on raw request handling capacity and response latency under increasing levels of concurrency. The tool was run on a dedicated host to prevent interference with the power measurement of the system under test and to simulate real external web traffic rather than relying on loopback interfaces. With wrk we used the extensibility of using Lua script to model some specific behavior which needed a higher load generation than JMeter. In this thesis, wrk was used with this command:

```
taskset -c 0-16 wrk -t17 -c17 -d20s -R$RATE -T1s --latency -H
 ↪  "Connection: Close" http://$IMPLEM
```

The command above launches `wrk` from the DeathStarBench suite with the following configuration: 17 threads (`-t17`), 17 concurrent TCP connections (`-c17`), a total test duration of 20 seconds (`-d20s`). The `-latency` option enables detailed latency measurements, while the HTTP header `Connection: Close` is added to each

request to ensure that the TCP connection is closed after each HTTP transaction.

To ensure consistent CPU core availability during the load generation, `taskset` was used to pin the `wrk` process to cores 0 through 16 (`-c 0-16`). This setup follows the same principle as described in Section 3.2, to isolate the load generation activity and avoid interference from other processes, resulting in more stable and reproducible performance measurements.

Apache JMeter [18], in contrast, is a more feature-rich and extensible testing tool designed for complex load scenarios. It supports a wide variety of protocols and test plans, allowing for more realistic user behavior simulations, such as session management, form submissions, and configurable think times, etc. JMeter was used to simulate more complex user interactions and to model more application level workloads, providing a complementary perspective to the wrk-based tests. Like wrk, JMeter was run on a separate host to ensure isolated energy measurements. In this thesis, Jmeter was used with this command:

```
JVM_ARGS="-Xms512m -Xmx76800m" taskset -c 0-16
↪  /opt/jmeter/bin/jmeter -n -t /login.jmx -Jrate=$RATE
↪  -Jtarget=$IMPLEM -l -Jsave_file="/result_jmeter.csv"
```

The command above launches `JMeter` using `taskset` to bind the process to CPU cores 0 through 16, following the same approach described previously to ensure consistent CPU core availability for load generation. The `JVM_ARGS` environment variable is set to `"-Xms512m -Xmx76800m"`, configuring the Java Virtual Machine (JVM) with an initial heap size of 512 MB and a maximum heap size of 76.8 GB. Increasing the heap size helps maintain stable load generation, especially under heavy concurrency, by reducing the likelihood of garbage collection interruptions during the test.

JMeter test script relies on a Thread Group which simulates the users, and in the thread group we can add several components like:

- **Samplers** to send HTTP requests to an endpoint (e.g., GET/POST/PUT to an API).
- **Listeners** which gather and visualize data from the test (like response times, throughput, error rates).
- **Timers** which are used, in our case, to control pacing and maintain a stable rate of requests (e.g., constant throughput timer).
- **Assertions** to validate that the responses meet expected conditions (status codes, response content).

- **Controllers** to manage flow logic (loops, conditions, transactions).
- **Pre-processors** and **Post-processors** to modify requests or extract data from responses dynamically.

The JMeter script is configured with 17 users, corresponding to 17 threads internally. Each thread operates independently and continuously sends requests to the target web application according to the specified rate parameter (`-Jrate=$RATE`). This one-to-one mapping between CPU cores and JMeter threads ensures efficient resource utilization and minimizes thread contention.

Both tools were used to generate workloads against the target web server implementations. Each test was run multiple times to account for variability.

3.4 Data Collection & Analysis

To run the experiment across the different parameters and collect the results into a dataset, the Network Performance Framework (NPF) [26] was used, a Python tool designed to execute Bash scripts with variable inputs and aggregate the resulting data into CSV format. An overview of a typical execution pipeline is provided in Figure 3.3. For each implementation under test, a corresponding NPF script was developed. These scripts varied the input rate (increasing exponentially) and cycled through the different implementations.

During the execution, the load generator is first invoked for each parameter combination to send requests to the respective implementation. Following each test run, performance metrics such as average request rate, average latency, 99th percentile latency, and throughput were collected from the load generator. Energy consumption data was then gathered using PowerAPI. To minimize cross-test interference, a sleep time was introduced between successive test executions.

Once all data had been collected via NPF, a separate Python script was executed to generate the related graphs. To assess whether the observed differences between implementations and configurations were statistically significant, hypothesis testing methods were used. Specifically, a one-way Analysis of Variance (ANOVA) was conducted with a significance level (α) of 0.05 to determine if any statistically meaningful differences existed between the implementations. The one-way ANOVA is a statistical test used to compare the means of independent groups to determine whether at least one group mean is significantly different from the others.

If the ANOVA test yielded a statistically significant result (indicating that not all group

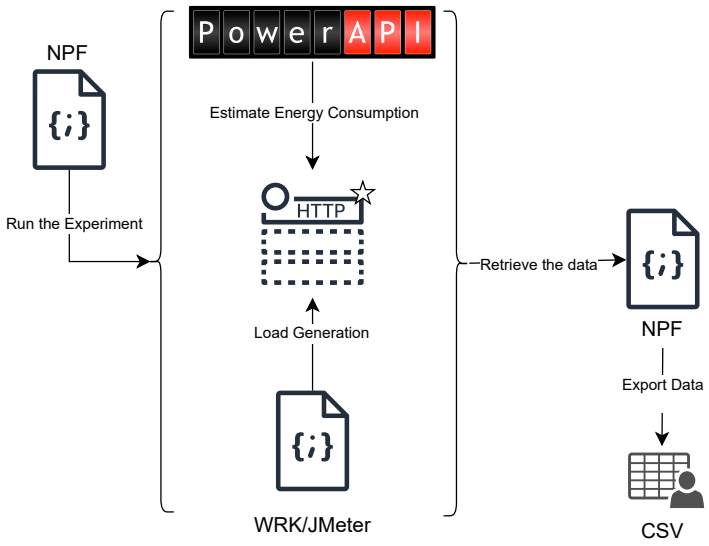


Figure 3.3: Overview of a typical data collection pipeline, with npf which run the experiments for all the different parameters, PowerAPI with estimate the energy consumption, wrk/JMeter which generates the http load and once again npf which retrieves the result into a CSV and a python script to analyze the resulting data.

means were equal) a more in depth analysis was conducted using pairwise t-tests. These t-tests compared the means of each possible pair of implementations for a given request rate, again using a significance level of 0.05. The use of statistical test allowed to draw more robust conclusions regarding the performance and energy efficiency trade-offs among the tested implementations.

3.5 Static Generation

In order to evaluate the effectiveness of static site generation across different web technologies. This section outlines the tools and processes used to generate static versions of each application.

3.5.1 WordPress

The static version of the WordPress site was generated using the Simply Static plugin [33]. Simply Static generates static (HTML) copies of all the WordPress pages. It works like a web crawler, starting at the home page of the application, and looks for links to other pages, working recursively. It also includes image, CSS, and JavaScript files, as well as

any other assets linked from the website.

3.5.2 Flask

To generate a static version of the Flask application, the Frozen-Flask tool was utilized [11]. This python script crawls the application by issuing requests to all accessible routes from the flask application and follows internal links to collect the corresponding static assets (HTML, CSS, JavaScript). For dynamic routes requiring parameters (e.g., /blog/id_1) a generator must be created and must explicitly yield all expected parameter values to ensure complete page generation, in the case of the example all the possibles blog page ID's. For flask application that relies on POST requests to modify the application or has the ability to have logged-in users, a more advanced setup was build for this thesis and is explained in Chapter 4.

Chapter 4

Contribution

The contribution of this thesis is the development of a hybrid architecture that effectively combines the advantages of both static and dynamic web application. The goal of this approach is to serve content while retaining the ability to handle dynamic operations such as POST requests, user authentication, and database modifications. And having a faster update for the user as a cache would do as when using cache the update to the application is only seen by the user when the cache is updated depending on the cache configuration.

This architecture has been realized using a combination of the web server to redirect the request and Flask with SQLAlchemy for the backend of the application and Flask-Frozen to be able to generate a static version of the site. The system differentiates between GET and POST requests at the web server level: static files are served directly for GET requests, while POST requests are proxied to the dynamic Flask backend. This allows dynamic features like content update and creation, user login without compromising the performance benefits of a static site.

To ensure that the static representation of the site remains up-to-date after dynamic changes (e.g., creating a new blog post or editing a post), the Flask application is configured to automatically trigger a rebuild of the static site whenever the database is modified. This is achieved using SQLAlchemy event listeners in combination with Flask-Frozen, as shown in the code snippet below Listing 1.

At the web server level, a redirection rule in NGINX ensures that GET requests are routed to the pre-built static version of the application, while all other requests are forwarded to the Flask backend Listing 2.

```python
def handle_post_change(mapper, connection, target):
    freezer.freeze()

event.listen(Post, "after_insert", handle_post_change)
event.listen(Post, "after_update", handle_post_change)
event.listen(Post, "after_delete", handle_post_change)
```

Listing 1: SQLAlchemy event listeners used to trigger static site regeneration when blog post data is modified.

```
location / {
    if ($request_method = GET) {
        rewrite ^/(.*)$ /build/$1 last;
    }

    proxy_pass http://127.0.0.1:9090;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Forwarded-Prefix /;
}
```

Listing 2: NGINX configuration for hybrid routing. GET request are routed to the static version of the site (/build) while the rest are routed to the flask application.

The end-to-end flow of a request in this hybrid architecture is illustrated below in the schema below Figure 4.1.

This hybrid design provides an efficient solution for building web applications that require dynamic interaction while retaining the scalability, speed, and SEO benefits of static sites. It demonstrates a novel yet practical approach to bridging the gap between modern static site generation and traditional dynamic web services.

A key distinction between the hybrid static-dynamic approach and traditional cache-based approaches resides in how updates are propagated to the end user.

In a typical cache-based system, dynamic content is generated on-the-fly by the server and then stored in a cache (e.g., using reverse proxies like Varnish or NGINX). Subsequent requests are served from the cache to reduce server load and improve response time. However, a major limitation of this approach is that updates to the application (such as new posts or database changes) are not reflected immediately. Instead, users continue to receive the stale cached version until the cache expires based on a predefined timeout or

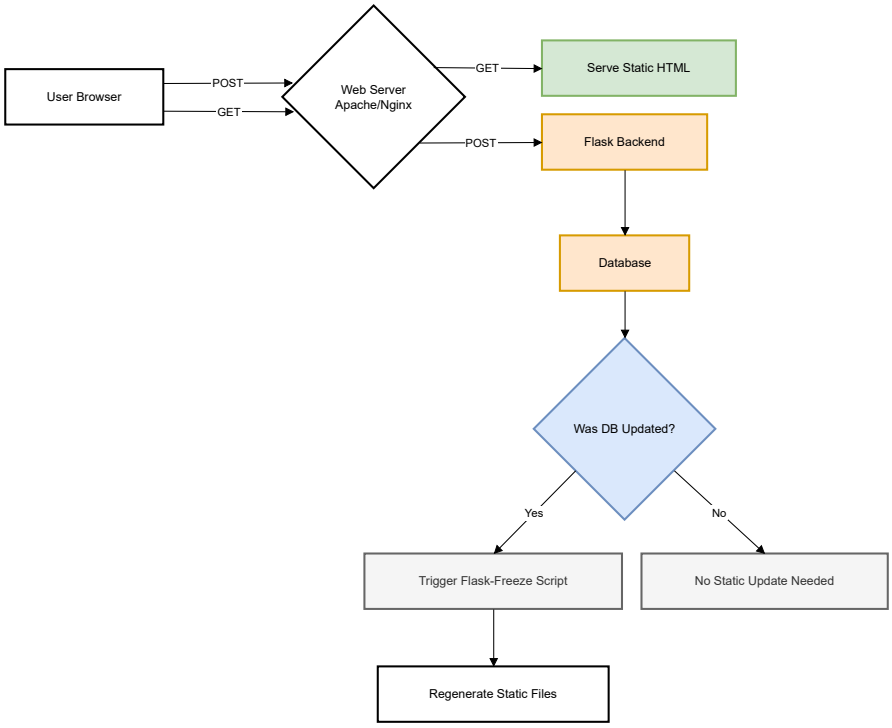


Figure 4.1: Hybrid architecture request flow: GET requests are routed to static files, while POST requests are handled by the dynamic Flask backend. Database updates trigger regeneration of the static site using Flask-Frozen.

is manually invalidated. This introduces a delay between when content is updated in the backend and when it becomes visible on the frontend.

By contrast, the hybrid static-dynamic model ensures immediate consistency between backend updates and what the user sees. When a POST request modifies the application's database (such as when a new blog post is created) the system rebuild the static files using Flask-Frozen. This rebuild reflects the new change in the regenerated HTML files. Consequently, the static version of the site is updated instantly, and users accessing the site through standard GET requests will see the most recent version without delay.

Chapter 5

Results

This section talks about the results of the study. First with a discussion about the experiment involving WordPress and its static version, followed by an analysis of the Flask experiments and its various implementations. Finally, the comparison between the energy efficiency of the two web servers used: Apache and Nginx.

To determine the impact of caching versus generating a static version of the website, a comparison of two caching strategies applied to the dynamic applications was done: one that stores cached responses in memory, and another that stores them on disk. This comparison was done for each of the application tested in this thesis.

For each test case, experiments were conducted using both Apache and NGINX web servers. However, since the results obtained from both servers were consistent across all test scenarios, this section will focus primarily on discussing the energy consumption measurements obtained from the Apache web server. An overview of the NGINX results is provided in Appendix B for reference.

Both web servers were configured with two caching strategies: one utilizing disk-based caching and the other employing in-memory (RAM) caching. During the experiments, it was observed that the estimated CPU energy consumption did not show significant statistical variation between the two caching strategies. As a result, the analysis presented here will focus on the disk-based caching method.

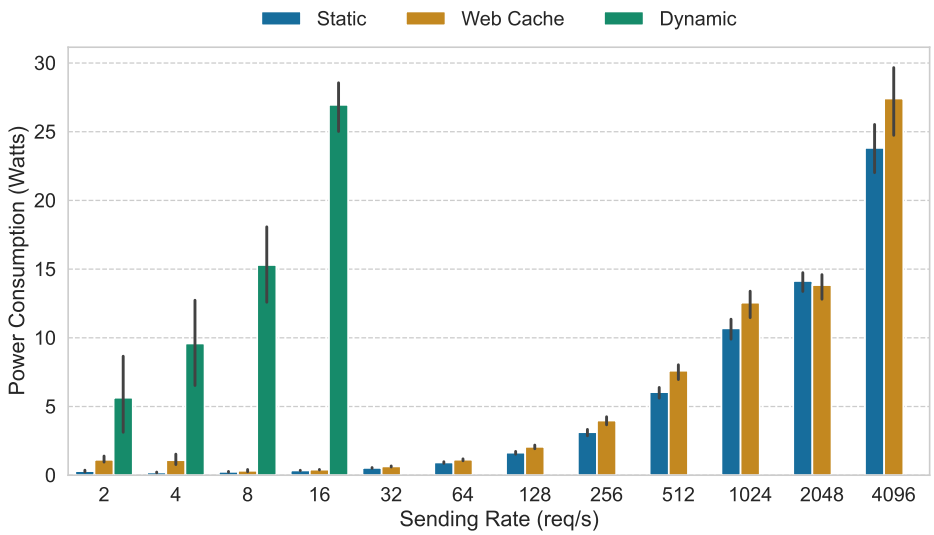


Figure 5.1: Comparison of energy consumption between WordPress application in dynamic, statically generated WordPress application, and with caching using the Apache web server. Experiments were conducted on the G5k Nancy Gros cluster. The Figure show the mean power usage (in watts) across increasing request rates up to the maximum for each configuration. Error bars indicate the 95% confidence interval.

5.1 WordPress

This section evaluates and compare the energy consumption of a WordPress web application and its static equivalent. WordPress is a widely used, free, and open-source content management system (CMS), powering over 43% of all websites globally [5]. It is developed in PHP and relies on a MySQL database to serve dynamic content.

The experiments were conducted using both Apache and Nginx web servers. A notable distinction between these web servers is how they handle PHP application. Apache supports native PHP integration through its built-in modules. In contrast, Nginx does not natively support PHP execution and requires the use of an external tool PHP-FPM (FastCGI Process Manager) to handle the execution of the PHP scripts. Therefore, PHP-FPM was employed when deploying WordPress with Nginx, while Apache uses the built-in modules. As explained before the results discussed will only be about the Apache web server, the results for the Nginx web server can be seen in the Appendix B.

5.1.1 Results

The results of our experiments are presented in Figure 5.1, which shows the energy consumption of the WordPress application under different configurations: fully dynamic, with caching (in-memory and on-disk), and as a static site. The experiments were conducted by increasing the request rate, the graphs show the mean power consumption in watts, with a 95% confidence intervals.

The results show that the dynamic WordPress configuration has the highest energy consumption across all request rates, peaking at an average of 27 watts at 16 requests per second (rps). In contrast, the cached versions and the static site showed significantly lower power usage, with an average reduction of over 60 times fewer watts compared to the dynamic version and having, at 16 requests per second (rps), less than 1 watt of power consumption and at 4096 requests per second (rps) an average of 20–25 watts. Statistical analysis using t-test ($\alpha = 0.05$) revealed no significant difference in power consumption between the static and cached configurations. However, the static version supported a significantly higher request throughput, handling up to 7350 rps, compared to a maximum of 5700 rps for the cached versions. The dynamic version maximum rate is 20 rps.

These findings indicate that static site generation provides the most energy-efficient and scalable solution for serving WordPress-based content. Caching, is a practical alternative when static generation is infeasible, as it follow the same energy consumption profile as the static version but can handle a lower maximum throughput. We also have to bear in mind that the power consumption is low because, as explained in the methodology (Chapter 3), only one CPU core was used for the web server.

5.2 Flask

In this section, we compare the energy consumption associated with deploying a static and a dynamic web application implemented in Python using the Flask framework. Flask is a lightweight and modular web framework written in Python that is particularly well-suited for building small to medium-sized web applications and RESTful APIs. It follows the WSGI (Web Server Gateway Interface) specification, which defines a standard interface between web servers and Python web applications or frameworks.

As for WordPress application, the implementations were tested using both Apache and Nginx web servers. A notable difference between the two setups is how they support WSGI applications. Apache integrates Python applications using the mod_wsgi module, allowing the WSGI interface to be configured directly within the web server. In contrast, Nginx does not provide native support for WSGI. Therefore, in the Nginx configuration,

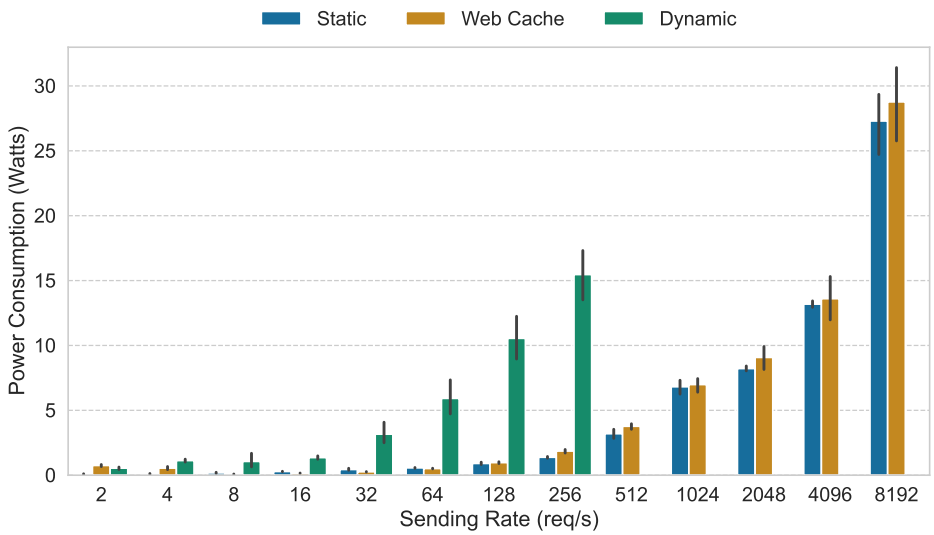


Figure 5.2: Comparison of energy consumption between the Flask application in dynamic, statically generated (Frozen-Flask), and with caching using the Apache web server. Experiments were conducted on the G5k Nancy Gros cluster. The Figure show the mean power usage (in watts) across increasing request rates up to the maximum for each configuration. Error bars indicate the 95% confidence interval.

gunicorn[13] a WSGI HTTP server for Python applications was used to interface with the Flask application. As explained before the results discussed will only be about the Apache web server, the results for the Nginx web server can be seen in the Appendix B.

With flask, we have 3 different test serving two different application, a simple test which only sends GET request to a basic blog application, another one which compare the different implementation with added POST requests on the basic blog application. And last, a test with another application which simulates an application which have a login.

5.2.1 Base Blog Application

The application used in this study is a minimalist blog application. It features a homepage that lists all available articles and dynamically serves individual pages for each blog article. The primary purpose of this application is to evaluate and compare the energy consumption of a simple dynamic blog, which is commonly implemented using platforms like WordPress but built here with Flask to add a comparison with the blog built with WordPress in the previous section.

Results

The results of the experiments are presented in Figure 5.2, which shows the energy consumption of the basic flask blog application under different configurations: fully dynamic, with caching (in-memory and on-disk), and as a static site. The experiments were conducted by progressively increasing the request rate and measuring the mean power consumption in watts, with a 95% confidence intervals.

The results show that the dynamic flask configuration resulted in the highest energy consumption across all request rates, peaking at 15 watts at 256 rps. In contrast, the cached versions and the static site showed significantly lower power usage, with an average reduction of over 9 times fewer watts compared to the dynamic version with a power consumption of around 2.5 watts at 256 rps. Statistical analysis using t-test ($\alpha = 0.05$) revealed no significant difference in power consumption between the static and cached configurations. The dynamic version maximum rate with Apache is 400 rps, compared to a max rate of, 10,000 rps for the two other configuration with a power consumption of 27 watts at 8192 rps.

Like the results with WordPress, these results confirm that static site provides the most energy efficient and scalable solution to serve Flask content. Caching, is a practical alternative when static generation is infeasible.

5.2.2 Post Blog Application

This experiment builds upon the Flask blog application introduced in the previous section by adding POST requests to simulate a more complex and interactive web application. Specifically, POST requests are used to create new blog posts, which are then rendered on the homepage and on a new individual article page.

The objective of this comparison is to evaluate the energy consumption of a dynamic web application capable of handling both GET and POST requests. Results are compared against a *static* version of the application, as well as with caching.

As explained in the contribution (Chapter 4) a fully static implementation is not feasible for this use case due to the requirement for user interaction, the ability to submit new posts and to view the resulting changes. To address this, the hybrid approach (called post_pass_through) was used, which uses Apache and Nginx abilities to redirect incoming requests. POST requests are forwarded to the Flask backend, while GET requests are redirected to pre-generated static files. To ensure changes from POST requests are reflected on the web page, flask frozen script is triggered right after each successful POST request. This script regenerates the necessary static files to reflect the new state of the blog.

Two testing strategies were employed to assess the energy impact of POST requests:

- Variable Load with Fixed POST Ratio: In this scenario, the total number of requests was incrementally increased while maintaining a constant ratio of 1% POST and 99% GET requests. The 1% POST ratio was selected to reflect a realistic update-to-view ratio for typical blog applications. This simulation was conducted using the wrk benchmarking tool, paired with a Lua script which with a random percentage send either a post request or a get request using the specified proportions.

- Fixed Load with Variable POST Ratio: Here, the total request rate was held constant at 256 requests per second, while the proportion of POST requests was gradually increased. This allowed for an analysis of how varying write operations affect overall energy consumption across different implementation strategies. The same wrk tool and Lua script were used in this test.

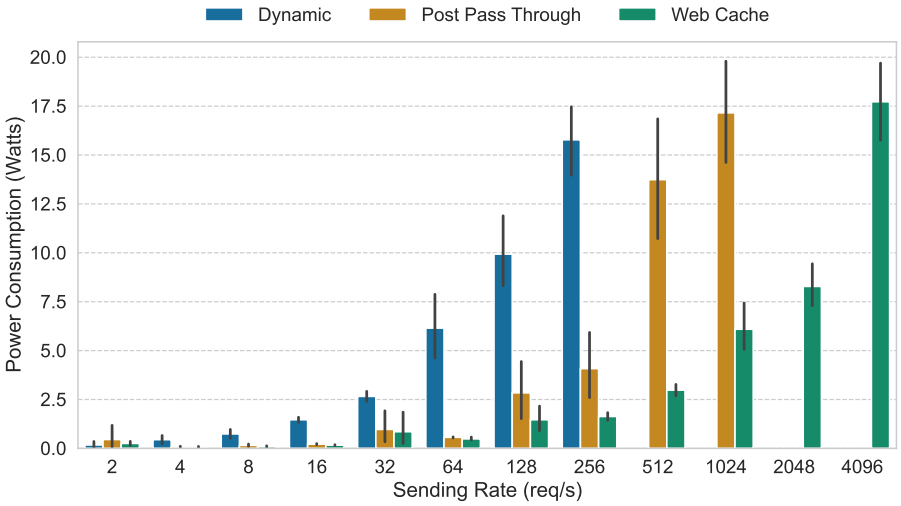
Results

Figure 5.3a shows the energy consumption of the Flask application with 1% of requests being POST request which create a new post on the application to show on the home page. Energy consumption vary across all implementation variants—fully dynamic, cached, and hybrid (which uses redirection and rebuilds the site using Flask-Frozen).

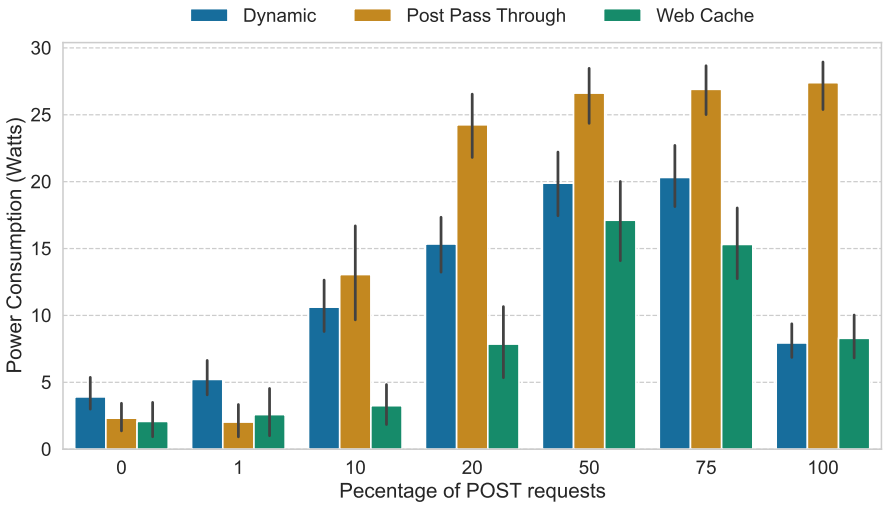
The results show that the dynamic implementation reaches a maximum throughput of 300 rps with a power consumption of 15 watts at 256 rps. The post_pass_through variant handle a maximum throughput of 1,200 rps with a power consumption of 16 watts at 1024 rps. Both caching-based version follows a similar pattern, reaching a peak rate of 5,000 rps, with a power consumption of 19 watts at 4,096 rps. With these results, we can see that for a typical application handling 1% of POST requests, the POST pass-through (hybrid approach) is a good alternative. It has a higher maximum throughput, consumes less power than the fully dynamic application, and avoids serving stale content, which can occur with caching. Stale content refers to outdated or no longer valid information that is served from a cache rather than being freshly generated or retrieved. This can be problematic in applications where content changes frequently or where users require up-to-date information. However, if serving stale content is acceptable for the application context, relying on caching remains the better alternative due to its simplicity and efficiency.

To provide a broader comparison, Figure 5.3 shows how energy consumption changes as the percentage of POST requests increases across implementations.

As the proportion of POST requests increases, the energy efficiency advantage of the post pas through approach diminishes. This is because each POST request rebuild the site



(a) Comparison of energy consumption between the Flask application with 1% POST request.



(b) Comparison of energy consumption between the Flask application with increasing POST request rate.

Figure 5.3: Comparison of energy consumption between the Flask application with POST requests in dynamic, statically generated (Frozen-Flask), and with caching using the Apache web server. Experiments were conducted on the G5k Nancy Gros cluster. The Figure show the mean power usage (in watts) across increasing request rates up to the maximum for each configuration. Error bars indicate the 95% confidence interval.

using Flask-Frozen, which is highly resource-intensive as it simulates requests to every possible page. Interestingly, the dynamic implementation consumes less energy at 100% POST requests, as a big energy cost comes from HTML generation and at full POST load when the page has a lot of data to show on the page. When the POST percentage is lower but still significant, the dynamic implementation becomes less efficient due to the overhead of rendering many huge pages.

The cache-based implementations remain solid alternatives across all POST request rates. They consistently consume less energy than the dynamic version, assuming some content staleness is acceptable. As expected, when the POST rate reaches 100%, caching provides no benefit, and both the cached and dynamic versions demonstrate similar energy usage, since the cache is never utilized.

5.2.3 Login Application

This part of the study explores a more advanced dynamic web application that adds user authentication. The application supports user login functionality, which could gives personalized content delivery to authenticated users, while unauthenticated users are served a static version of the site. This setup allows for a performance and energy consumption comparison between dynamic and static content delivery when using features which are user specific, such as those found on news platforms.

As explained in the contribution (Chapter 4) a full static approach is not feasible for applications which requires authentication. Therefore, a hybrid method, similar to the one used when testing with POST requests, was implemented. When the request has a session cookie, thus when the user is logged in to the website, all GET requests are router to the dynamic Flask application. In contrast, unauthenticated users receive the built static content. Other request types, such as POST, are always handled by the Flask backend.

To evaluate and compare the energy consumption in scenarios with user login, the Apache JMeter tool was used. The test flow, used for the test case is shown in Figure 5.4, includes:

1. Accessing the homepage without a session cookie
2. navigating to the login page
3. Submitting login credentials via a POST request
4. Returning to the homepage as an authenticated user

This approach allows for consistent evaluation of energy consumption across implemen-

tations using full dynamic routing, caching, and hybrid redirection mechanisms.

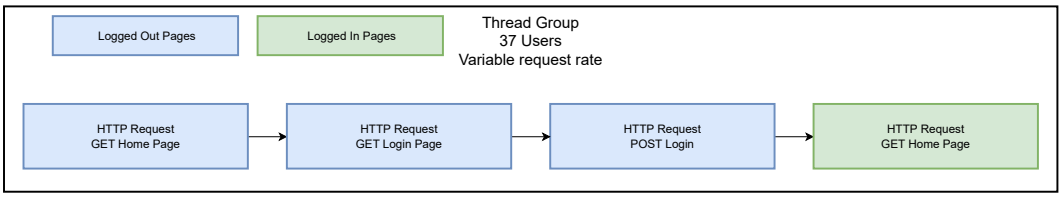


Figure 5.4: JMeter test plan simulating user login flow: accessing homepage, logging in, and re-accessing personalized content.

Another test case was conducted to compare energy consumption as the percentage of logged-in users varies, and to show a simulation for a website like Wikipedia where most users browse without logging in.

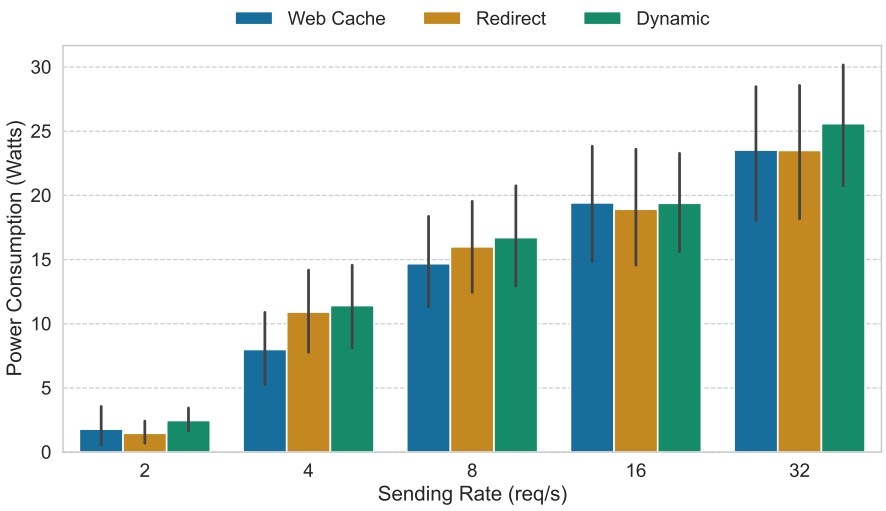
Results

Figure 5.5a shows the energy consumption of the Flask login application under increasing request rates. The energy consumption is nearly identical among all implementation variants: fully dynamic, cached, and hybrid (using redirection). This is expected, as the login test plan involves every user authenticating. Since caching and redirection only apply to unauthenticated access, their impact is minimal when 100% of users are logged in. Having for the max rate of 32 rps an average of around 25 watts in power consumption.

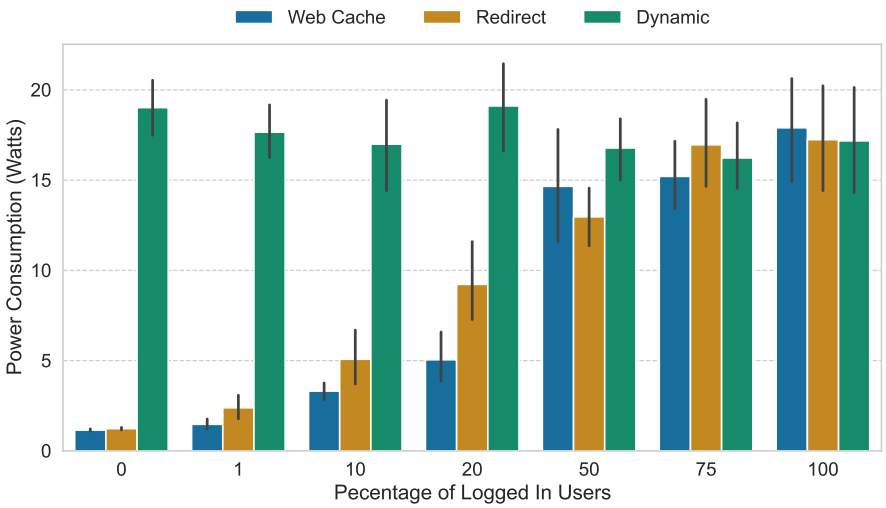
The maximum achievable request rate in this test was limited due to constraints in Apache JMeter's load generation. To address this, a second experiment was conducted using wrk, allowing for higher throughput and to simulate variation in the proportions of logged-in users.

Figure 5.5b shows the energy consumption as the percentage of authenticated users increases, with a constant request rate of 64 rps. As expected, higher login proportion lead to greater energy use in the hybrid and cached implementations, since a larger portion of requests requires dynamic processing. However, the benefits of using the cached version or the hybrid method remain significant when the percentage of logged-in users is low to moderate. For instance, when 0% of users are logged in, the dynamic application consumes around 18 watts, whereas the cached and hybrid versions use only 1 watt.

The energy-saving advantage diminishes as the login percentage increases. By the time 75% of users are authenticated, energy usage across all implementations becomes comparable at around 15–17 watts for all implementations, which shows that the optimizations



(a) Comparison of energy consumption as the request rate increases.



(b) Comparison of energy consumption as the percentage of logged-in users increases, with a rate of 64 rps.

Figure 5.5: Comparison of energy consumption of the flask application with logged-in users across different Flask application with login: dynamic, statically generated (Frozen-Flask) using redirection, and with caching on the Apache web server. Experiments were conducted on the G5k Nancy Gros cluster. The Figure show the mean power usage (in watts) across increasing request rates up to the maximum for each configuration. Error bars indicate the 95% confidence interval.

lose their benefit when most users require dynamic content. It is verified with the statistical test that when the percentage of users is 75% or more, there is no significant difference between all the implementation. Still, for sites where a majority of users remain unauthenticated, such as Wikipedia, using caching or hybrid redirection can lead to important energy savings.

The results confirm that using static content delivery based on user authentication or using caching can significantly reduce energy consumption when the proportion of user authenticated is lower than 75%.

5.3 Nginx and Apache comparison

This section compares the energy consumption of the two web servers (Apache and Nginx) used in this thesis. The comparison reuses results from previous tests, but to ensure a fair evaluation, only the static version of each application is considered. This is because the dynamic and cached versions relied on different tools for each web servers to serve the dynamic application, making a direct comparison unreliable.

Two test cases were conducted: one using a static version of the WordPress application, and the other using a static version of a simple Flask-based blog. These cases are useful for comparison because they differ significantly in file size. The static WordPress page served a larger file of approximately 83,040 Bytes, while the Flask application served a smaller file of around 3,157 Bytes.

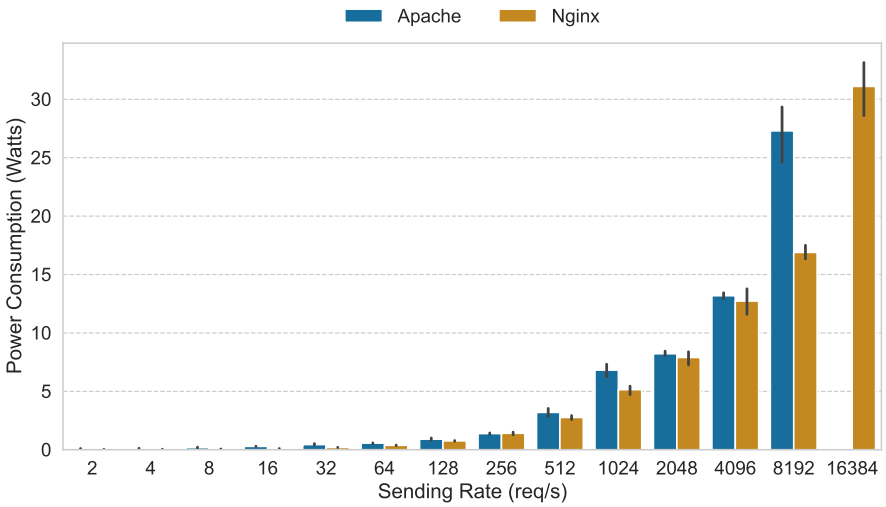
Results

Figure 5.6 presents the measured energy consumption for serving static content using both Apache and Nginx web servers, across two applications: a minimal Flask-based static site (serving a smaller file) and a WordPress static site (serving a larger file).

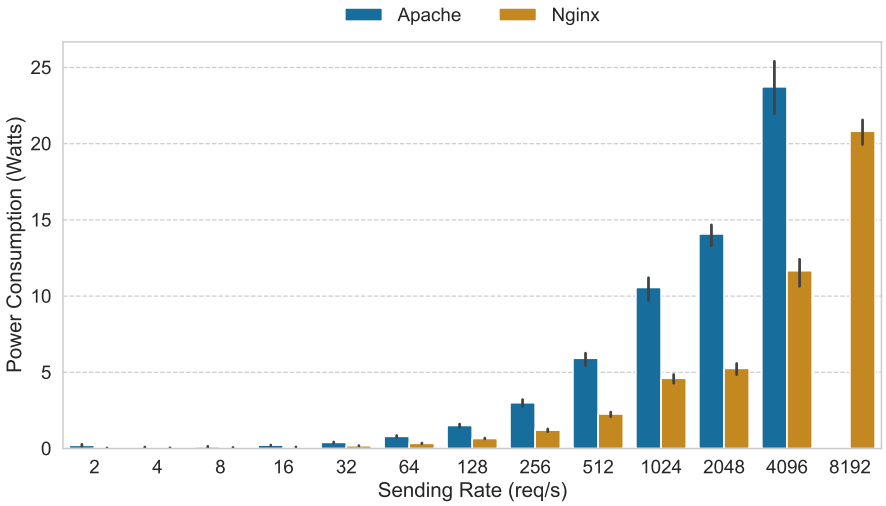
Across all request rates, Nginx demonstrated significant lower energy consumption than Apache, as confirmed by using independent t-tests at a significance level of $\alpha = 0.05$.

This trend is consistent in both test cases, and is more pronounced under higher load and with a bigger file (WordPress test case). In the WordPress scenario (Figure 5.6b), Apache consumed on average 1.9 times more power than Nginx. Similarly, in the Flask scenario (Figure 5.6a), Apache used approximately 2.8 times more power on average. We can therefor see that Apache generally consume more than nginx, but when handling a bigger file size the difference is more pronounced.

Nginx also handled significantly higher request loads compared to Apache before reach-



(a) Energy consumption of the Flask static application with the two web servers.



(b) Energy consumption of the WordPress static application with the two web servers.

Figure 5.6: Energy consumption (in watts) of Apache and Nginx when serving static files from the Flask and WordPress applications. Experiments were conducted on the G5k Nancy Gros cluster. The Figure display average power usage across increasing request rates, with error bars representing the 95% confidence interval.

ing its limit. For the Flask static site, Apache reached a maximum stable throughput of 10,000 requests per second, consuming 25 watts at 8,192 requests/sec, while Nginx handled up to 20,000 requests per second, consuming 30 watts at 16,384 requests/sec. In the WordPress case, Apache peaked at 6,500 requests per second (24 watts at 4,096), whereas Nginx supported up to 13,000 requests per second (20 watts at 8,192).

These results demonstrate that using different web servers can not only reduce energy consumption when serving static content, but also increase the maximal throughput and has a higher impact the bigger the file is and higher the throughput is.

For the other experiments we verify the results stay consistent for the other experiments, results from the other experiments with the nginx sever can be viewed in the appendix (Appendix B).

Chapter 6

Threats to validity

This section will give an analysis of the possible threats to validity related to this study. The threats have been build from the guidelines for measuring energy consumption made by Ardito et al. [2].

6.1 External Validity

External validity concerns the generalizability of our results beyond the specific test conditions used in our study.

6.1.1 Interaction of Treatment and Selection

Different hosting environments and server hardware can produce varying power consumption results and maximum request differences due to difference in energy efficiency, processor architectures, etc. To mitigate this threat, we provide the test system specifications, including hardware configurations. To guarantee reproducibility, users must employ the same software versions and carry out tests under controlled conditions. In addition, all experiments are carried out on a wired network to avoid variability introduced by Wi-Fi fluctuations.

6.1.2 Temporal Validity

Software versions evolve gradually over time, which can lead to variations in energy consumption. Just as hardware improvements can enhance efficiency, so too can software, changing observed trends in energy consumption. To address this threat, we document

the software versions used in the study. However, future studies may need to re-evaluate the results as new versions of web servers, frameworks, etc. become available.

6.2 Internal Validity

Internal validity refers to the degree to which our experiment design ensures that the observed effects are uniquely attributable to the configurations tested.

Our study is based on a power estimation model, PowerAPI, which gathers data from Running Average Power Limit (RAPL) sensors and Linux performance monitoring tools. PowerAPI estimates the energy on a per-process basis, so that the execution of other process on the server does not induce a change in the energy measurement of the configuration under test.

Additionally, energy sampling is not expected to be significantly influenced by the OS or other running processes. The test environment is configured to limit background activity, and multiple iterations of each benchmark are performed to reduce variability.

6.3 Construct Validity

Construct validity assesses whether the measurements accurately reflect actual power consumption differences. In this study, the power estimation is derived from a model rather than direct hardware-based power measurements. Therefore, the accuracy of the results is conditioned on the precision of the model used. PowerAPI's accuracy has been validated against a hardware powermeter, showing a margin of error ranging from 0.5% to 3% [25], which ensures a reliable estimation of energy consumption.

6.4 Conclusion Validity

Conclusion validity relates to the reliability of statistical analyses in supporting our findings.

To mitigate threats to conclusion validity, multiple iterations of each test are executed to reduce the impact of outliers. Statistical methods such as Analysis of Variance (ANOVA) and independents T-Test are employed to assess the significance of observed differences. These measures ensure that our conclusions are based on robust statistical evidence and not random fluctuations in power measurements.

Chapter 7

Conclusion

This thesis set out to evaluate and compare the energy consumption of different approaches to hosting web applications, fully dynamic, cached, and static deployments using both WordPress and Flask frameworks deployed on two web servers, Apache and Nginx. Through various experiments and benchmarking under varying request loads and application features, the study has showed that the choice of hosting strategy (static, dynamic, caching) and web servers has a great impact on energy efficiency and on the maximum possible rate for each application.

The results confirm that using a static version of the website, consistently yields the lowest energy consumption and the highest throughput across all scenarios, making it the most sustainable approach when applicable. For instance, the Flask static site achieved a maximum throughput of 10,000 requests per second using Nginx, consuming only 27 watts at 16,384 requests/sec. In contrast, the dynamic Flask configuration peaked at only 400 requests/sec with Apache, consuming 15 watts at just 256 requests/sec. Similarly, WordPress in static supported up to 7,350 requests/sec, while the dynamic version only managed 20 requests/sec, with a power consumption peaking at 27 watts.

Caching mechanisms, especially in-memory caching, offer a strong alternative when static deployment is not feasible, providing significant energy savings over dynamic rendering while maintaining great performance. For example, the cached Flask application handled up to 10,000 requests/sec, consuming 2.5 watts at 256 requests/sec, compared to 15 watts for the dynamic version (a 9x reduction in energy usage).

However, caching efficiency diminishes with increasing dynamic user interactions, such as POST requests or authenticated sessions. In the Flask login test, where all users authenticated, power consumption for all implementations converged around 25 watts at

32 requests/sec. But when 0% of users were logged in, the cached and hybrid versions used just 1 watt, compared to 18 watts for the dynamic version. Similar results are observed for the hybrid approach when handling POST requests and login mechanisms. It is a viable alternative in scenarios with low user interaction, particularly when immediate dynamic updates are required from the user's perspective. Otherwise, employing caching strategies within a dynamic application offers a more efficient solution.

The dynamic configurations, while functionally necessary in many real-world applications, consistently exhibited the highest energy usage and the lowest scalability, underlining the importance of minimizing real-time computation when possible.

The choice of web server also showed significant influence on performance and energy usage. Across all tests, Nginx outperformed Apache. In the WordPress scenario, Apache consumed 1.9× more power, and in the Flask scenario, 2.8× more. For example, in Flask static tests, Apache capped out at 10,000 requests/sec at 25 watts, while Nginx sustained 20,000 requests/sec at 30 watts—double the throughput with only 20% more power.

In conclusion, developers, aiming to reduce the environmental impact of web services should prioritize static content delivery where possible, leverage caching strategies appropriately, and consider an energy efficient web server.

Chapter 8

Future work

While this thesis provides an initial exploration into the differences between the energy consumption of static and dynamic web applications, several possible directions for further research still remains.

One possible direction is the extension of this study to include a broader range of web frameworks and technologies. Comparing additional frameworks such as Django, Ruby on Rails, or Node.js and their static site generation counterparts would allow for a more comprehensive understanding of how different architectures and programming languages impact energy efficiency.

Another important expansion would be to evaluate the influence of additional web servers beyond Apache and Nginx. Testing lightweight servers such as Caddy, lighthhtp, or highly optimized enterprise solutions like LiteSpeed, could offer further insights into how the choice of web server software affect the overall energy footprint of web applications.

Another possible expansion would be to evaluate the influence of using a stack of phones to serve content and comparing the results of energy consumption between classic web servers, NAS, and ultra-low-power board.

Furthermore, introducing more complex web applications into the experiments would help bridge the gap between simple blogs and modern production scale websites. Future studies could include applications that heavily rely on external APIs, as well as websites following microservice based architectures, which distribute functionality across multiple interconnected services hosted on several machines. These setups are increasingly common in today's web ecosystems and may show significantly different energy usage patterns compared to monolithic architectures used in this thesis.

Finally, integrating real-world traffic patterns, user interactions, and scaling scenarios would make future experiments even more representative of operational environments. This would enable more accurate modeling of energy consumption under different load conditions and usage profiles.

By pursuing these extensions, future research can build a more detailed understanding of sustainable web development practices.

Bibliography

[1] Tedis Agolli, Lori Pollock, and James Clause. Investigating decreasing energy usage in mobile apps via indistinguishable color changes. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 30–34. IEEE, 2017.

[2] Luca Ardito, Riccardo Coppola, Maurizio Morisio, and Marco Torchiano. Methodological guidelines for measuring energy consumption of software applications. *Scientific Programming*, 2019(1):5284645, 2019.

[3] Simon Arys and Romain Carlier. *Energy-aware scheduling for ISA-heterogeneous clusters: from process-level power estimations to cluster-wide energy efficiency*. PhD thesis, UCLouvain, 2024.

[4] Lotfi Belkhir and Ahmed Elmeligi. Assessing ict global emissions footprint: Trends to 2040 & recommendations. *Journal of cleaner production*, 177:448–463, 2018.

[5] Usage statistics and market shares of content management systems. `https://w3techs.com/technologies/overview/content_management`. Accessed: 2025-04-25.

[6] Felicia de Mander and Wilhelm Gren. *Comparison of Energy Usage and Response Time for Web Frameworks*. PhD thesis, Blekinge Institute of Technology, 2023.

[7] Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy conservation policies for web servers. In *4th USENIX Symposium on Internet Technologies and Systems (USITS 03)*, 2003.

[8] Brad Everman and Ziliang Zong. GreenWeb: Hosting High-Load Websites Using Low-Power Servers. In *2018 Ninth International Green and Sustainable Computing Conference (IGSC)*, pages 1–6, Pittsburgh, PA, USA, October 2018. IEEE.

[9] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. Smartwatts: Self-calibrating software-defined power meter for containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 479–488. IEEE, 2020.

[10] Flask. https://flask.palletsprojects.com/en/stable/. Accessed: 2025-04-12.

[11] Frozen flask. https://frozen-flask.readthedocs.io/en/stable/index.html. Accessed: 2025-04-12.

[12] Grid 5000. https://www.grid5000.fr/w/Grid5000:Home. Accessed: 2025-04-12.

[13] gunicorn. https://gunicorn.org/. Accessed: 2025-04-25.

[14] Hardware performance counter (hwpc) sensor. https://powerapi.org/reference/sensors/hwpc-sensor/. Accessed: 2025-04-02.

[15] Ralph Hintemann and Simon Hinterholzer. Energy consumption of data centers worldwide. *Business, Computer Science (ICT4S)*, 2019.

[16] Stefan Huber, Lukas Demetz, and Michael Felderer. A comparative study on the energy consumption of progressive web apps. *Information Systems*, 108:102017, 2022.

[17] Robert Istrate, Victor Tulus, Robert N Grass, Laurent Vanbever, Wendelin J Stark, and Gonzalo Guillén-Gosálbez. The environmental sustainability of digital content consumption. *Nature Communications*, 15(1):3724, 2024.

[18] Apache jmeter. https://jmeter.apache.org/. Accessed: 2025-04-12.

[19] Joulemeter. https://www.microsoft.com/en-us/research/project/joulemeter-computational-energy-measurement-and-optimizatic Accessed: 2025-05-25.

[20] Maja H Kirkeby, John P Gallagher, and Bent Thomsen. An approach to estimating energy consumption of web-based it systems. In *CERCIRAS*, 2021.

[21] Martijn Koot and Fons Wijnhoven. Usage impact on data center electricity needs: A system dynamic forecasting model. *Applied Energy*, 291:116798, 2021.

[22] Jorg Lenhardt, Kai Chen, and Wolfram Schiffmann. Energy-Efficient Web Server Load Balancing. *IEEE Systems Journal*, 11(2):878–888, June 2017. Number: 2.

[23] Irene Manotas, Cagri Sahin, James Clause, Lori Pollock, and Kristina Winbladh. Investigating the impacts of web servers on web application energy usage. In *2013 2nd International Workshop on Green and Sustainable Software (GREENS)*, pages 16–23. IEEE, 2013.

[24] Antti P. Miettinen and Jukka K. Nurminen. Analysis of the Energy Consumption of JavaScript Based Mobile Web Applications. In Periklis Chatzimisios, Christos Verikoukis, Ignacio Santamaría, Massimiliano Laddomada, and Oliver Hoffmann, editors, *Mobile Lightweight Wireless Systems*, pages 124–135, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[25] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. A review of energy measurement approaches. *ACM SIGOPS Operating Systems Review*, 47(3):42–49, November 2013. Number: 3.

[26] Npf. `https://github.com/tbarbette/npf`. Accessed: 2025-04-12.

[27] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA, October 2017. Association for Computing Machinery.

[28] Olivier Philippot, Alain Anglade, and Thierry Leboucq. Characterization of the energy consumption of websites: Impact of website implementation on resource consumption:. In *ICT for Sustainability 2014 (ICT4S-14)*, Stockholm, Sweden, 2014.

[29] Powerapi. `https://powerapi.org/`. Accessed: 2025-04-02.

[30] powertop. `https://github.com/fenrus75/powertop`. Accessed: 2025-05-25.

[31] Chris Preist, Daniel Schien, and Paul Shabajee. Evaluating sustainable interaction design of digital services: The case of youtube. In *Proceedings of the 2019 CHI conference on human factors in computing systems*, pages 1–12, 2019.

[32] Varun Sapra and Abram Hindle. Web servers energy efficiency under http/2. *PeerJ Preprints*, 4:e2027v1, 2016.

[33] Wordpress simply static plugin. `https://github.com/Simply-Static/simply-static`. Accessed: 2025-04-25.

[34] Smartwatts formula. `https://powerapi.org/reference/formulas/smartwatts/`. Accessed: 2025-04-02.

[35] stress-ng. `https://github.com/ColinIanKing/stress-ng`. Accessed: 2025-04-25.

[36] Mario Tomiša, Marin Milković, and Marko Čačić. Performance evaluation of dynamic and static wordpress-based websites. In *2019 23rd International Computer Science and Engineering Conference (ICSEC)*, pages 321–324. IEEE, 2019.

[37] Max Van Hasselt, Kevin Huijzendveld, Nienke Noort, Sasja De Ruijter, Tanjina Islam, and Ivano Malavolta. Comparing the Energy Efficiency of WebAssembly and JavaScript in Web Applications on Android Mobile Devices. In *The International Conference on Evaluation and Assessment in Software Engineering 2022*, pages 140–149, Gothenburg Sweden, June 2022. ACM.

[38] Benoy Varghese, Niklas Carlsson, Guillaume Jourjon, Anirban Mahanti, and Prashant Shenoy. Greening web servers: A case for ultra low-power web servers. In *International Green Computing Conference*, pages 1–8, November 2014.

[39] Wordpress. `https://wordpress.com/`. Accessed: 2025-04-12.

[40] wrk 2. `https://github.com/delimitrou/DeathStarBench/blob/master/wrk2/README.md`. Accessed: 2025-04-12.

Appendices

Appendix A

Configuration Files

This section provides the necessary configuration files and website data to replicate the experimental setup and execute the experiments.

A.1 PowerAPI configurations

This section describes and give the configuration used by PowerAPI sensor and formula.

Listing 3 presents the configuration of the PowerAPI HwPC sensor. This sensor collects hardware performance counters every 1000 ms, using the default set of events which works on the G5k Gros instances. The collected data is stored in a locally hosted MongoDB instance.

```
{
  "name": "sensor",
  "verbose": true,
  "frequency": 1000,
  "output": {
    "type": "mongodb",
    "uri": "mongodb://127.0.0.1",
    "database": "db_sensor",
    "collection": "report"
  },
  "system": {
    "rapl": {
      "events": ["RAPL_ENERGY_PKG"],
      "monitoring_type": "MONITOR_ONE_CPU_PER_SOCKET"
    },
    "msr": {
      "events": ["TSC", "APERF", "MPERF"]
    }
  },
  "container": {
    "core": {
      "events": [
        "CPU_CLK_UNHALTED:REF_P",
        "CPU_CLK_UNHALTED:THREAD_P",
        "LLC_MISSES",
        "INSTRUCTIONS_RETIRED"
      ]
    }
  }
}
```

Listing 3: Configuration file for the HwPC sensor

Listing 4 shows the configuration of the PowerAPI SmartWatts formula. It processes the data collected by the HwPC sensor from the MongoDB instance and stores the computed results in an InfluxDB instance. The configuration uses the CPU characteristics, including the thermal design power (TDP) and base frequency, while disabling DRAM power consumption estimation.

```
{
  "verbose": true,
  "stream": true,
  "input": {
    "puller": {
      "model": "HWPCReport",
      "type": "mongodb",
      "uri": "mongodb://127.0.0.1",
      "db": "db_sensor",
      "collection": "report"
    }
  },
  "output": {
    "pusher_power": {
      "tags": "socket",
      "model": "PowerReport",
      "type": "influxdb2",
      "uri": "http://127.0.0.1",
      "port": 8086,
      "db": "BUCKET_NAME",
      "org": "ORG_NAME",
      "token": "ADMIN_TOKEN"
    }
  },
  "cpu-base-freq": 2200,
  "cpu-tdp": 125,
  "cpu-error-threshold": 2.0,
  "sensor-reports-frequency": 1000,
  "disable-dram-formula": true
}

```

Listing 4: Configuration file for the SmartWatts formula

A.2 NPF Scripts

This section provide an example of a npf script used in this thesis to run the experiment and collect the data from the experiments.

```
%config
result_append={POWER}
accept_zero={POWER}
%variables
RATE=[2*32768]
IMPLEM={flaskStatic.local:static,flaskCache.local:web_cache,
    flaskMemCache.local:web_mem_cache,flaskDynamic.local:dynamic}
%late_variables
DUR=20s
TIMEOUT=1s
THREADS=17
CONNECTION=17
%script
start_time=$(date +%s)
taskset -c 0-16 wrk -t$THREADS -c$CONNECTION -d$DUR -R$RATE \
    -T$TIMEOUT --latency \
    -H "Connection: Close" http://$IMPLEM &> output
end_time=$(date +%s)

cat output | grep "Transfer/sec" \
    | awk '{printf "RESULT-THROUGHPUT %s\n", $2}' | head -n 1
cat output | grep "Requests/sec" \
    | awk '{printf "RESULT-REQUEST %s\n", $2}' | head -n 1
cat output | grep "Latency" \
    | awk '{printf "RESULT-LATENCY %s\n", $2}' | head -n 1
cat output | grep "99.000%" \
    | awk '{printf "RESULT-LAT99 %s\n", $2}' | head -n 1

ORG="ORG_NAME"
BUCKET="BUCKET_NAME"
DB_TOKEN="ADMIN_TOKEN"
curl -s --request POST -m 3 \
  --url 'http://influx/api/v2/query?org='$ORG'&bucket='$BUCKET'' \
  --header 'Accept: application/csv' \
  --header 'Authorization: Token '$DB_TOKEN'' \
  --header 'Content-Type: application/vnd.flux' \
  --header 'content-type: text/plain' \
  --data 'from(bucket: "'$BUCKET'")
  |> range(start: '$start_time', stop: '$end_time')
  |> filter(fn: (r) => r["_measurement"] == "power_consumption")
  |> filter(fn: (r) => r["target"] == "/webgroup.slice/apache2")
  |> yield(name: "mean")' | tail -n+2 \
    | awk -F, '{printf "RESULT-POWER %s\n", $7}' | head -n-1
sleep 4s
```

Listing 5: NPF example scripts for the flask application running for flask, static, cache in disk, and cache in memory for the Apache web server

A.3 Website Configuration file

For the different configuration of the website and the web servers, you can access those on the GitHub for this master thesis `https://forge.uclouvain.be/ArnaudJungers/web-server-energy-comparison`

Appendix B

Nginx Results

B.1 WordPress

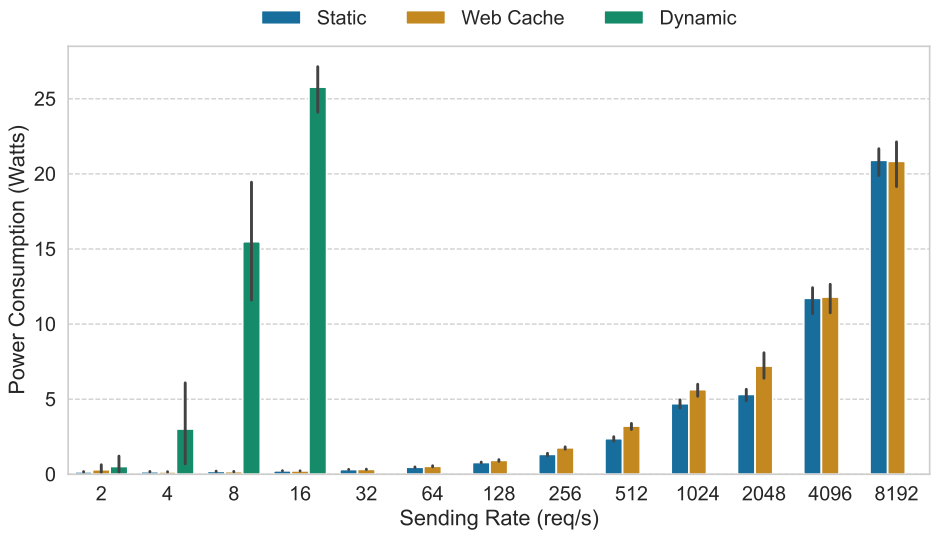


Figure B.1: Comparison of energy consumption between WordPress application in dynamic, statically generated WordPress application, and with caching using the Nginx web server. Experiments were conducted on the G5k Nancy Gros cluster. The Figure show the mean power usage (in watts) across increasing request rates up to the maximum for each configuration. Error bars indicate the 95% confidence interval.

B.2 Flask Base application

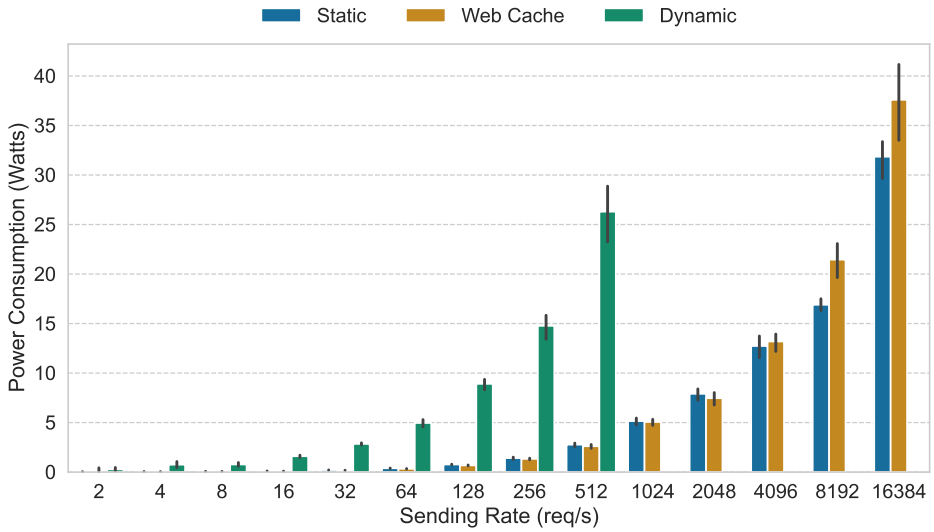
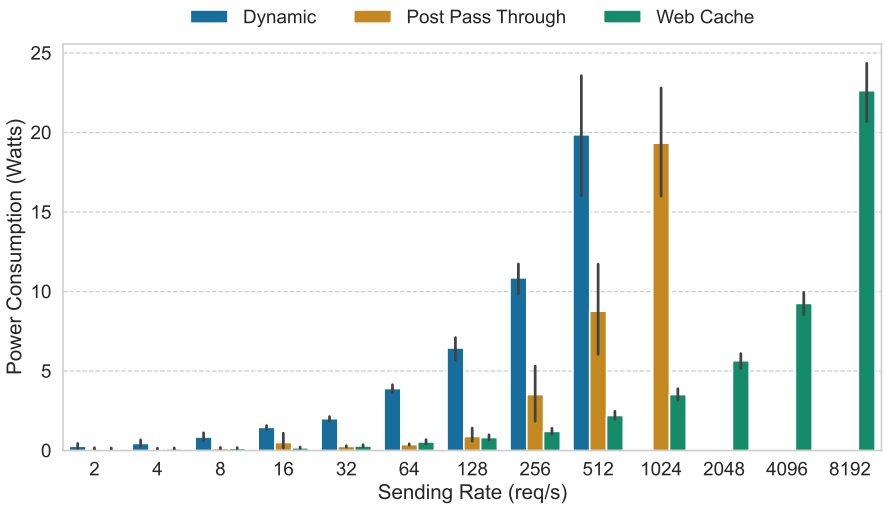
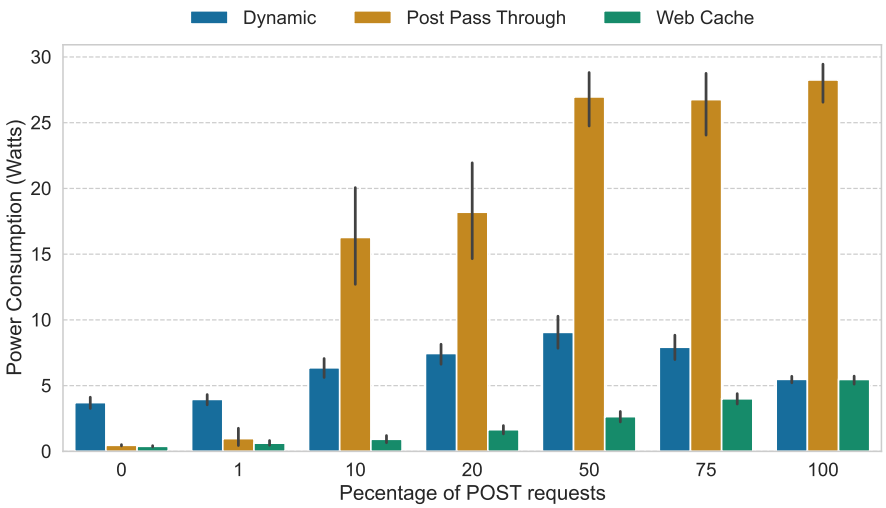


Figure B.2: Comparison of energy consumption between the Flask application in dynamic, statically generated (Frozen-Flask), and with caching using the Nginx web server. Experiments were conducted on the G5k Nancy Gros cluster. The Figure show the mean power usage (in watts) across increasing request rates up to the maximum for each configuration. Error bars indicate the 95% confidence interval.

B.3 Flask Post



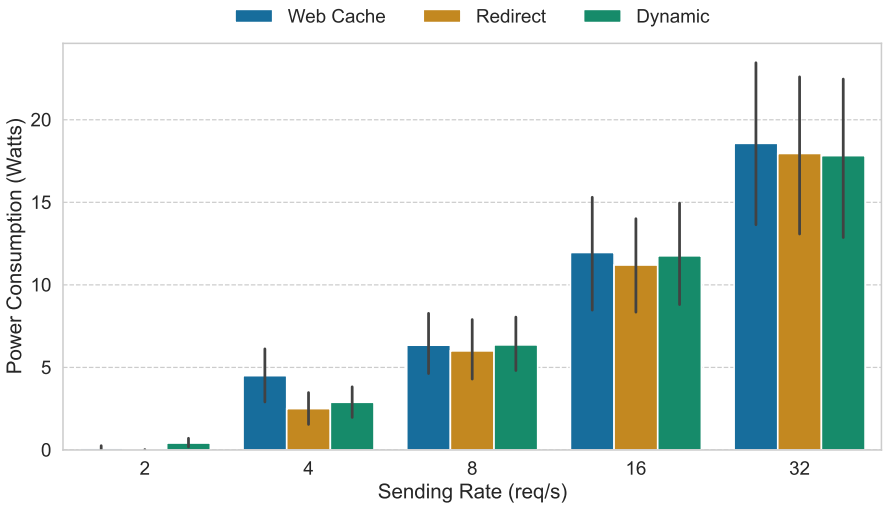
(a) Comparison of energy consumption between the Flask application with 1% POST request.



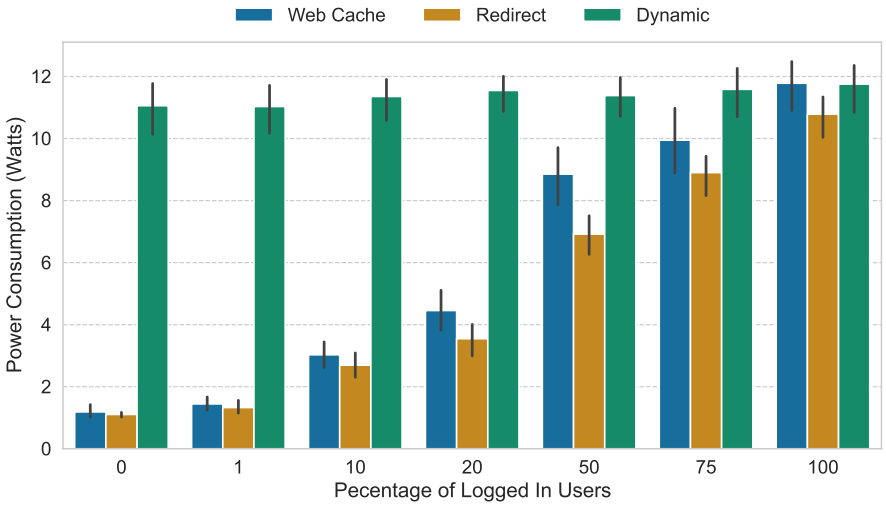
(b) Comparison of energy consumption between the Flask application with increasing POST request rate.

Figure B.3: Comparison of energy consumption between the Flask application with POST requests in dynamic, statically generated (Frozen-Flask), and with caching using the Nginx web server. Experiments were conducted on the G5k Nancy Gros cluster. The Figure show the mean power usage (in watts) across increasing request rates up to the maximum for each configuration. Error bars indicate the 95% confidence interval.

B.4 Flask Login



(a) Comparison of energy consumption as the request rate increases.



(b) Comparison of energy consumption as the percentage of logged-in users increases, with a rate of 64 rps.

Figure B.4: Comparison of energy consumption of the flask application with logged-in users across different Flask application with login: dynamic, statically generated (Frozen-Flask) using redirection, and with caching on the nginx web server. Experiments were conducted on the G5k Nancy Gros cluster. The Figure show the mean power usage (in watts) across increasing request rates up to the maximum for each configuration. Error bars indicate the 95% confidence interval.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**