SUMMARY

Dette kandidatspeciale benchmarker et udvalg af udbredte og populære konfigurationer af full-stack frameworks og sammenligner deres relative energieffektivitet. Specialet starter med at analysere eksisterende akademiske artikler, der bl.a. tester frameworks samt måler energiforbruget af andre web-relaterede teknologier. Dette leder til en problemformulering med tilhørende forskningsspørgsmål, der skal besvares.

For at beslutte hvilke frameworks som skal benchmarkes, konsulteres en liste af de 100 mest besøgte hjemmesider. Hver af disse hjemmesider tjekkes for, om de benytter et full-stack framework. Hvis ikke dette er tilfældet, fjernes de fra listen. Baseret på dette overblik samt undersøgelser af de individuelle frameworks' popularitet blandt udviklere, udvælges seks forskellige frameworks til benchmarken. Udover at notere deres frameworks, kategoriseres de individuelle hjemmesider baseret på deres indhold. For at sikre at testsiderne repræsenterer de mest besøgte hjemmesider, designes de således at alle de tidligere identificerede kategorier er repræsenteret på mindst én af deres fem undersider. Baseret på dette design implementeres identiske testsider i hvert framework.

For at måle energiforbruget af testsidens server, udvides et automatisk benchmarkværktøj, der i forvejen understøtter at måle klient-side energiforbrug samt båndbredde igennem Gecko Profileren. Udvidelsen benytter vores værktøj der er i stand til at rapportere processpecifikke energiforbrugsmålinger der fremskaffes vha. Recount, en del af macOS's XNU-kernel. Gecko Profileren benytter også Recount som kilde til sine målinger, hvilket gør det muligt at sammenligne klient- og serversidemålinger direkte.

For at sikre, at benchmarken udføres videnskabeligt og at resultaterne er statistisk signifikante, gør vi brug af flere statistiske analysemetoder. For at bestemme antallet af benchmarkiterationer benyttes Cochrans test. For at validere vores brug af ikke-parametriske tests bruger vi Kolmogorov-Smirnov-testen på vores data, som fortæller om dataene er normalfordelt. Idet vores data ikke er normalfordelt, skal vi bruge ikke-parametriske tests. Til sammenligning af de forskellige frameworks gør vi brug af Mann-Whitney U-testen, der tillader os at sammenligne dem parvis. For at undgå type I-fejl korrigerer vi p-værdierne med Holm-Bonferroni-metoden. Resultatet af dette er et overblik over hvilke frameworks der bruger statistisk signifikant mere strøm end andre.

Det er vigtigt at vores benchmarkresultater kan genskabes. Derfor beskriver vi i specialet, hvordan vi opsætter vores testmiljø. Dette inkluderer både, hvilke teknologier vi har brugt, hvilken computer vi har brugt til at udføre testen, og i hvilken rækkefølge de individuelle testsider kører. Vores benchmark kører i 60 iterationer, hvilket tager omkring 120 timer.

Resultaterne viser at energiforbruget på klientsiden er markant højere end energiforbruget på serversiden. Forskellene i målinger på klientsiden er relativt små. Den højeste forskel er på 82 %, og i en af testene er der kun 12 ud af 42 sammenligninger som er statistisk signifikante. På serversiden er forskellene i målingerne meget større. Den største forskel er på 2.856 %, og testene har som minimum 36 ud af 42 forskelle som er statistisk signifikante. Der er ingen åbenlys rangering af testsider, men vi kan stadig drage visse konklusioner.

`astro-htmx` er den eneste testside som rangerer over middel i alle tests på tværs af alle målinger, så længe man ignorerer målingerne, hvor der er for lille forskel til at bevise en statistisk signifikant forskel.

`ruby-rails-hotwire` og `ruby-rails-htmx` har mere blandede resultater, men der kan argumenteres for, at de er de næstbedste testsider. De rangerer især godt i energiforbrug på serversiden.

`nextjs` er den eneste testside som rangerer under middel i alle tests bortset fra én. Den er især dårlig i båndbreddemålingerne.

Hvis man laver et estimat på hvor meget energi der er brugt for en bestemt mængde båndbredde, kan man samle alle tre målinger til en samlet måling for hver test. I denne måling er det stadig `astro-htmx` der rangerer bedst og `nextjs` der rangerer værst, men den næstbedste testside er `django-htmx`, fordi den bruger relativt lidt båndbredde sammenlignet med `ruby-rails-hotwire` og `ruby-rails-htmx`.



Master Thesis w/ Department of Computer Science
Software, Master of Science (MSC) in Engineering
Selma Lagerløfs vej 300
9220, Aalborg
https://www.cs.aau.dk/

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Comparing Energy Consumption of Full-Stack Web Frameworks With a Novel Test Site and an Automated Benchmark

Theme:
Programming Technology

Project Period:
Spring 2026 - 4th Candidate Semester
(10th Semester)

Project Group:
cs-26-pt-10-01

Participants:
Albert Hald Christensen
Anton Juul-Naber
Bastian Brix Hansen

Supervisors:
Bent Thomsen
Lone Leth Thomsen

Page Numbers: 51

Date of Completion:
June 6, 2026

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Preface:

This paper is the product of a master's thesis project by three students from Aalborg University. The project spanned the period from 02-02-2026 to 08-06-2026.

All of the project artifacts can be found on GitHub in the organization *FrameworkBenchFullStack-RepPack*: https://github.com/FrameworkBench-RepPack. It contains a repository for each of the test sites, the database, the expanded benchmark runner, the expanded benchmark runner results parser, and the new process-measuring tool. Additionally, a replication package has been created, which includes a step-by-step guide to replicate the benchmark.

AI based tools have been used in several ways to support the creation of this project. Grammarly was used to improve punctuation and general sentence structure, while LLMs were used to get feedback on code, answer general questions, review the paper itself, and discover new literature. It should be noted that LLM-generated text has not been used directly in the paper.

We want to express our sincere gratitude to our supervisors, Lone Leth Thomsen and Bent Thomsen, for their invaluable guidance and support throughout the preliminary project and this master's thesis.

Comparing Energy Consumption of Full-Stack Web Frameworks With a Novel Test Site and an Automated Benchmark

Albert Christensen*, Anton Juul-Naber*, and Bastian Hansen*

*Department of Computer Science, Aalborg University

***Abstract*—This paper benchmarks a selection of widely used and popular full-stack frameworks and compares their relative energy efficiency. It presents an extension to a benchmark runner that automatically carries out predetermined interactions in Firefox using Selenium, while collecting client-side energy and bandwidth measurements using the Gecko Profiler. Server-side energy consumption measurements are gathered by querying Apple Recount, which is the same source used by the Gecko Profiler. Recount provides per-process results by attributing energy consumption at every thread context-switch. Consequently, the noise introduced by system and background processes is minimized, making the relative measurements more accurate. By measuring both client-side energy, server-side energy, and bandwidth consumption, the benchmark runner provides a comprehensive overview of the frameworks' consumption. A test site is designed that represents the most visited websites. It is implemented in seven different framework configurations. A total of six test cases is carried out on each test site implementation to challenge the full-stack frameworks in various tasks. The results show that the client side deviates up to 82 % in energy consumption, and that the server side deviates up to 2856 %, meaning that there is a potential to save energy based on the framework choice. The paper concludes that there is a statistically significant difference in the energy consumption, and that the frameworks Astro and Ruby on Rails perform the best, while Next.js performs the worst.**

I. Introduction

As global energy demands increase, the principles of green software engineering are being applied across all domains, including web development [1]. Given that the average internet user spends approximately 3,230 hours online annually, evaluating and reducing website energy consumption is a logical step toward lowering the overall carbon footprint of the Information and Communication Technology (ICT) sector [2].

Recent literature mostly evaluates websites based on traditional web metrics such as rendering speed and accessibility [3, 4]. There remains a notable gap in studies measuring the combined energy consumption of web applications and their underlying technologies. Without empirical data on these metrics, developers lack the necessary insights to make environmentally conscious architectural decisions when choosing technology for their websites.

To address this gap, this paper benchmarks seven full-stack framework configurations selected based on their use among the most visited 100 websites and developer popularity surveys. Using a standardized test site implemented in all seven configurations, their energy efficiency is evaluated while performing identical interaction sequences. To capture the complete operational lifecycle of a website, this study measures energy consumption on both the client and server sides, providing a comprehensive overview of the website's energy footprint. The goal of this paper is to measure the energy consumption of the most widely used full-stack frameworks and to enable the consideration of energy consumption when choosing a framework.

The main contributions of the paper are:

- Benchmark Results: Data that developers can utilize alongside traditional performance metrics when selecting a framework for their projects.
- An extended automated benchmark runner: A tool designed to programmatically perform predefined test cases while measuring client-side and server-side energy consumption, enabling replicable future benchmarking.
- A process-measuring tool that gathers process-isolated energy consumption measurements for a given program.
- Standardized Framework Implementations: A uniform

test site implemented across seven full-stack framework configurations, providing reusable sites for future benchmarking research.

Section II outlines the current state-of-the-art in benchmarking web technologies and specifies a problem formulation with accompanying Research Questions (RQs) based on the related works. Section III determines which frameworks will be tested, and which Object-Relational Mapping (ORM) will be used for the frameworks that do not ship with one by default. Section IV details how the benchmarked test site is designed based on an analysis of the most visited websites. It explains each subpage on the site and the test cases associated with them. Section V outlines the applied methodology by detailing the statistical analysis, how the benchmark is automated and performed, and how measurements are gathered. To improve replicability, the section then details the test environment and changes made to ensure stability, followed by a description of the order in which the tests are performed. Section VI examines the benchmark results and presents findings from comparing the energy and bandwidth consumption of each framework. Section VII covers the threats to validity and how these might influence the results. In Section VIII, the results are discussed. The energy consumption corresponding to the frameworks' bandwidth usage is also estimated and used to evaluate their combined consumption. The results are then compared to findings in recent literature. Based on the energy consumption results, the section provides guidelines on framework choices. Lastly, it details future research directions that further explore the energy consumption of full-stack frameworks and could provide further insights into the obtained results. Section IX concludes the paper by reflecting on the RQs and findings.

II. Related Work

Full-stack frameworks provide a uniform developer experience for managing client-side and server-side logic of web applications, associated network requests, and database integrations. They often describe themselves as batteries-included, meaning that they ship with a majority of the tools needed for building enterprise websites, such as routing tools, database management tools, and authentication tools. Despite their popularity, research comparing the energy consumption of web frameworks appears, at the time of writing, to remain limited to client-side front-end web frameworks [5].

A. Evaluating Client-Side Performance of Web Frameworks

The performance of web frameworks has historically been measured by metrics like Total Blocking Time (TBT) and Time to Interactive (TTI), where the focus has been more on usability than energy consumption. However, attempts have been made to attribute these metrics to the consumption of a website [3]. In 2025, [6] explored various techniques for measuring the energy consumption of websites. As part of their study, they examined a Gecko Profiler-based approach by comparing the energy consumption of the same website implemented using two different front-end frameworks. Their findings suggested that the energy consumption varied across frameworks. However, a key threat to validity was their reliance on manual operation of the profiler and the interaction script. Based on their recommendation of using CPU-based measurements while accommodating this limitation, [7] measured and compared the client-side energy consumption of the most popular and promising front-end web frameworks. The study found a statistically significant difference in energy consumption among different frameworks. Additionally, it found that the most popular frameworks were generally the least efficient. In 2021, [8] used custom micro-benchmarks to evaluate the rendering performance of popular front-end web frameworks. The study found that the best-performing rendering strategies are the ones that minimize the Document Object Model (DOM) size by utilizing techniques such as compile-time optimizations and reactive programming models. A limitation of these studies is their exclusive focus on client-side metrics, despite the popularity of full-stack frameworks [5].

B. Evaluating Full-stack Frameworks

Research relating to full-stack frameworks has primarily focused on examining various usability aspects. In 2022, [9] examined the most common vulnerabilities found in web applications and how the built-in security features of popular full-stack frameworks safeguard against these. In 2018, [10] scraped questions from Stack Overflow related to two major full-stack frameworks called Django and Laravel, attempting to determine the most common issues and help framework developers understand the needs of their users.

Performance-related studies have also evaluated the back-end API capabilities of various full-stack frameworks. In 2025, [11] examined the API performance of four popular back-end frameworks: Spring Boot, ASP.NET Core, Express.js, and

Django. The study found that ASP.NET Core achieved the lowest average latency, while Django exhibited the highest average latency. In the same year, [12] tested the same frameworks as well as Laravel, and observed similar results, with ASP.NET Core achieving the shortest response times, while Django and Laravel exhibited the highest.

To assess the energy efficiency of web application back-ends, [13] proposed REST-Q, a framework for assessing the energy consumption of web applications. It contains a client that sends HTTP requests to the back-end while measuring its energy consumption using Scaphandre and PowerAPI. The HTTP requests trigger database queries sourced from well-known database benchmarks. The tool runs experiments on web application servers to test the impact of their different software configurations. A shortcoming of the tool is that it does not account for client-side energy consumption, as it measures only the back end. Client-side energy consumption is significantly influenced by the utilized front-end technologies, making it an important factor to consider [7]. Benchmarking the energy efficiency of specific full-stack frameworks has historically been limited to the back-end, with implementations of the same Application Programming Interface (API) across different frameworks [14].

C. Measuring Energy Consumption on the Web

There has been an increasing interest in measuring the energy consumption of web browsers and websites. In 2012, [15] compared the energy consumption of different popular websites and found that downloading and parsing Cascading Style Sheets (CSS) and JavaScript (JS) has a large impact on the energy consumption. In 2020, [16] used Intel Running Average Power Limit (RAPL) to compare the energy consumption of Google Chrome and Mozilla Firefox, finding that Chrome was the most efficient browser overall. Another study from the same year measured the battery consumption of mobile browsers using digital power meters and found that ad-blocking browsers increase efficiency by 20-40 % [17]. In 2025, [18] examined the energy consumption of desktop browsers by continuously polling the current battery discharge rate from the Linux power supply subsystem, and similarly found that ad-blocking browsers were the most efficient, aligning with earlier findings.

Measuring back-end energy consumption of a web application was in [19], achieved using the Dynatrace Carbon Impact App. Based on this app's original paper, it relies exclusively on estimations based on hardware utilization and CPU TDP, which, according to their own results, do not accurately predict actual energy usage [20]. The previously mentioned framework, REST-Q, deploys Scaphandre [21], a tool for attributing energy consumption to specific processes. Recent literature has presented improved alternatives, namely EnergAt [22] and Metrion [23]. EnergAt improved the distribution estimation in dual-socket server settings, while Metrion introduced a model that incorporates frequency scaling.

Finally, in [7], the Gecko Profiler is utilized to measure client-side energy consumption due to its ability to report process-isolated measurements on Apple Silicon-powered devices. In this configuration, the Gecko Profiler uses Apple Recount, an alternative to Metrion and Scaphandre, which directly attributes power to individual processes in the kernel instead of estimating a distribution.

D. Problem Formulation

Full-stack frameworks remain popular in the industry [24]. These frameworks are used by websites to support the large volumes of traffic and content [11]. With the increased focus on transitioning to greener software and the lack of insight into the energy consumption of full-stack frameworks, it is not possible for developers to consider full-stack framework choices based on energy consumption. This leads to the problem formulation:

How can a benchmark and measurement tool be developed to automatically perform replicable benchmarks while measuring both client and server energy consumption of full-stack framework-based test sites consisting of commonly used web elements? How can statistical analysis be applied to the collected measurements to compare the different frameworks based on their energy consumption? Based on the results, can we provide insight and recommendations on the technology choices when choosing frameworks for a project?

To answer this formulation, we need to answer the following RQs:

RQ1: Is there a statistically significant difference in the client-side and server-side energy consumption of web applications built with different full-stack frameworks?

RQ2: Are there any observable patterns in the energy consumption difference?

To answer those RQs, we must also ask the following RQs:

RQ3: How can the energy consumption of both the front-end and back-end of a website be measured?

RQ4: How can a test site be designed and implemented to be representative of the most visited full-stack framework-based websites?

RQ5: How can test cases be designed to include web components commonly found on the most visited websites built using full-stack frameworks?

III. Choosing Frameworks to Benchmark

A full-stack framework is a software tool that provides the front-end and back-end tools needed to build a website [25]. To create dynamic pages with data binding, most full-stack frameworks include a templating library. The back-end tools often include a routing engine to handle Server-Side Rendering (SSR) and URL-mapping out of the box, an ORM which facilitates type-safe database communication, and business logic support. This removes the need to either develop or include a lot of dependencies by providing everything packaged as a complete ecosystem.

Some of these frameworks use the Model View Controller (MVC) pattern, which was originally popularized by Django in 2005, and subsequently became the main pattern for similar frameworks [26]. However, this began to change in 2016 as component-based architectures and file-based routing became more common in full-stack frameworks. Examples of such frameworks include Next.js, which uses React, and ASP.NET, which introduced Razor pages that could be used instead of the MVC pattern [27, 28]. The goal of a full-stack framework is to speed up the development process by making it safer and easier to maintain, so that developers can focus on building the product rather than on the tools [25].

A. Chosen Frameworks

To determine the frameworks that must be tested, we examine a list of the most visited 100 websites sourced from [7]. Each website in the list is visited and analyzed to discover which of them use a full-stack framework. Out of the 100 most visited websites, 71 of them use one. The final list of websites can be found in Appendix A. Frameworks used by these 71 sites reflect the most popular server-side web frameworks used in the industry, apart from Laravel, which none of the websites use, instead opting for another PHP framework; Symfony, with much lower popularity [24]. An overview of the frameworks utilized by the 71 websites can be seen in Table XVIII. The list of benchmarked frameworks can be found in Table I.

Framework	Description
ASP.NET Core	Framework created in C# by Microsoft. Has both MVC and component-based architecture.
Spring Boot	MVC framework created in Java
Django	MVC framework created in Python
Ruby on Rails	MVC framework created in Ruby
Next.js	JavaScript framework that uses the component-based front-end framework React
Astro	JavaScript Framework that allows for any component-based front-end framework

TABLE I: The frameworks included in the benchmark.

The frameworks are all full-stack server-side frameworks except for Astro. Despite being more front-end-focused, it is included because it represents a newer approach to using frameworks. Astro handles routing and layout much like Next.js with components and file-based routing. It is built for websites like blogs and archives that are more static and content-heavy, and is one of the most energy-efficient web frameworks [7, 29]. By also testing Astro, we aim to discover whether a newer framework, which is more focused on the client side, consumes less energy than the larger server-side focused full-stack frameworks.

B. Rendering the Website

To better compare the frameworks, we use the same rendering techniques for each of the frameworks. In cases where content needs to be changed dynamically on the site, such as a dynamic list, the modern approach is for the server to render the Hypertext Markup Language (HTML) and then swap the outdated content with the new, instead of re-rendering the whole page or completely reloading it. This eliminates the need to re-render static content, such as the navigation bar, footer, or text on a page, when dynamic elements change. This approach improves speed and responsiveness [30].

Of the chosen frameworks, only Next.js and Ruby on Rails come pre-installed with libraries that support this. Next.js uses React Server Components, which allow individual components to be rendered on the server [31]. Ruby on Rails has Hotwire, which uses Turbo to split the page into components that can be rendered on the server [32]. The other frameworks do not ship with such a library. To remove any potential bias from using different front-end libraries, HTMX is used. HTMX is chosen because it is the most popular framework-agnostic library [33], and is a library that provides extensions

TS ORM	npm downloads (2026-05-14 - 2026-05-20)
Prisma	10,640,532 [38]
Drizzle ORM	8,107,951 [39]
Kysely	5,060,069 [40]
TypeORM	3,857,225 [41]
Knex.js	3,459,943 [42]
MikroORM	531,083 [43]

TABLE II: TS ORMs and their popularity.

to HTML-attributes that enable Asynchronous JavaScript and XML (AJAX) [34]. Its functionality mimics that of React Server Components and Hotwire, meaning that HTML can be reused or swapped without needing full-page reloads, which results in the same behavior across all test sites. In contrast to Next.js, Ruby on Rails supports using different front-end libraries. To compare the efficiency of Ruby on Rails with the HTMX-based implementations, we therefore also implement a Ruby on Rails test site with HTMX.

C. Object-Relational Mapping (ORM)

All full-stack frameworks listed in Table I provide their own ORM, except for Next.js and Astro. To enable type-safe database interactions, we therefore have to use a third-party ORM. The selection of TypeScript (TS) ORMs will be based on a comparison of their popularity and their performance. By inspecting the Node.js package manager npm, multiple TS compatible ORMs are discovered: Prisma, Drizzle ORM, TypeORM, Kysely, Knex, MicroORM. The popularity of the ORMs is determined based on their weekly npm downloads, which can be seen in Table II. Prisma and Drizzle are more popular compared to the remaining options. The runtime performance of Prisma and Drizzle has been compared in multiple benchmarks [35, 36]. In both of these, Drizzle scored better than Prisma. In another benchmark, in which all of the previously mentioned ORMs are tested, Drizzle performs the best in 11 of 14 tests and outperforms Prisma in all of them [37]. Due to Drizzle's popularity and the fact that it consistently outperforms the other ORMs, it will be used for interacting with the database in the Next.js and Astro-based test sites.

IV. Test Site

As mentioned in Section III-A, the top 100 most visited websites are inspected. This is done to answer RQ 4, which questions how a test site that is representative of the most visited websites can be designed. While inspecting each website, it is determined whether they use a full-stack framework. Websites that do not use full-stack frameworks are not further inspected. The remaining websites are then categorized based on their content. Each of the discovered categories contain a combination of web elements that are unique to it. Based on this list of categories and their web elements, a test site is designed. To make the website as representative as possible, multiple subpages are used to represent the different categories. The full analysis and design are discussed in Appendix A.

Based on the above analysis, the test site design is determined to consist of five pages. An overview of these pages can be seen in Table III.

Page	Path	Description
Home	/	A rich landing page with content from all other pages
Static	/static-1	A page containing static text
Static	/static-2	A page containing static text
Live	/live	A page which streams live data from the server
List	/list	A page which contains a filterable list

TABLE III: An overview of the test site pages.

While the different pages contain unique main content, they all include similar headers and footers. These headers and footers contain navigation links to the remaining subpages.

The different implementations can be found on GitHub [44–50].

A. Test Cases

To answer RQ 5 about designing test cases, the findings from inspecting the top 100 most visited websites are used. Based on the identified website categories, the previously discussed subpages are designed to include commonly used web components such as lists, filters, static text, and streamed live content.

The test cases are designed around these subpages to evaluate how the full-stack frameworks handle interactions with the commonly found elements on each page. Six test cases are created to test different performance aspects of these interactions. Each of them is implemented and executed using the benchmark runner, which is documented in Appendix B.B.

The `static` test case loads the `static-1` page and closes it immediately thereafter. This presents the smallest possible workload on the server, only requesting that it render static text and send it to the client.

The `live` test case loads the `live` page and waits 20 seconds before closing the page. This tests how efficiently the server can send out periodical updates, and how efficiently the website can receive and render the updates. The live data is sent from the server and consists of three numbers, which are deterministically swapped with three new numbers every second.

The `list` test case loads the `list` page. This tests how efficiently the server can perform a default query for 100 people and render the result in a list.

The `list-interact` test case loads the `list` page and interacts with a filtering form, filtering the category of people, their age, and name. This tests how efficiently the server can deliver partial updates to the list, and how efficiently the website can receive and render the lists. When the test interacts with the form, the client sends a request to the server, which queries the database for the needed persons. The server then generates the new list and sends it to the client.

The `home-interact` test case loads the `home` page, and interacts with the filtering form. This tests how efficiently the server and website handle a more complex page, which is simultaneously receiving live updates and receiving new lists. Like the `list-interact` case, the list in the `home-interact` gets filtered using a form, all while the live data, like in `live`, is continuously updated.

The `navigate` test case performs a navigation across all pages. The order is `home`, `static-1`, `static-2`, `live`, `list`, and then the same order in reverse. This tests how efficiently the web page can navigate to new pages.

B. Test Site Interactions

Some front-end frameworks attempt to predict user intent to improve the perceived responsiveness. An example of such an optimization is Next.js's use of speculative prefetching triggered by mouse movement to make navigation seem faster [51]. To ensure that the frameworks are able to utilize such optimizations during the benchmark, interactions are not performed instantaneously. Instead, delays and mouse movement are used to simulate real user behavior.

When interacting with form input fields, real user behavior is simulated by slowly moving the mouse towards the element in a straight line, hovering it for a short while, and finally clicking the element. Similarly, each keyboard input is delayed to simulate a human writing on a keyboard.

V. Method

A. Statistical Analysis

To answer RQ 1, we must determine if there is a significant difference between the frameworks' consumption. The statistical methods used are inspired by [7] due to similarities in the data and the performed analysis. We determine the number of test iterations, check for normality in the data, check whether the data contains statistically significantly different energy measurement results, and then compare the test results pair-wise.

The required number of iterations is determined using Cochran's formula [52]. It uses the standard deviation from a preliminary test, as well as the industry-standard significance level of 95 % and 2 % margin of error [53]. To select a standard deviation, we choose the largest standard deviation of the client and the server, to cover the worst-case scenario.

To determine which statistical methods apply to our data, we assess normality using a Kolmogorov-Smirnov test [54]. We expect the data to be non-normal based on the findings in [7].

A Kruskal-Wallis test is performed on each test case to determine whether there is any statistically significant difference in the results [55]. These tests check whether the results contain at least one test site result that is statistically significantly different, justifying a pairwise comparison of the energy measurements. For the pairwise comparison, we use one-sided Mann-Whitney U tests on the eligible data [56]. To reduce the number of Type I errors, the P-values are corrected using Holm-Bonferroni [57]. The result is a pairwise comparison that for every framework combination in each test case shows whether they use statistically significantly different amounts of energy.

The above-mentioned analysis is carried out using R-scripts, which can be found in the GitHub repository [58].

B. Performing the Benchmark

The benchmark will be performed using an extended version of the automated benchmark runner from [7], which uses the Gecko Profiler and Apple Recount for measuring the energy consumption. By automating the benchmark process, we avoid the threats to validity mentioned in [6] related to manual interaction with the Gecko Profiler and manually starting an interaction script. To automate Gecko Profiler and browser interaction, Selenium is used [59]. Selenium is chosen for its ability to execute code in the browser's chrome

(the browser's own User Interface (UI)), which is required for programmatically interacting with the Gecko Profiler. In addition to avoiding the threats to validity, the benchmark runner also improves the replicability of the benchmark. The benchmark runner is further described in Appendix B and its implementation can be found on GitHub [60].

C. Gathering Measurements

To compare the frameworks, three metrics are measured: bandwidth, client-side energy, and server-side energy. The Gecko Profiler measures client-side energy consumption by continuously polling system interfaces for energy consumption information. Since Firefox launches each tab as a separate process, the profiler is able to report measurements for tabs individually. Because the Gecko Profiler is limited to measuring Gecko instances, a different approach is required for measuring the server-side, hence the need for an extension to the benchmark runner. To measure the server-side energy consumption, a process-measuring tool is developed. Similar to the Gecko Profiler, the process-measuring tool manages the server process and measures its energy consumption. To enable direct comparison between client-side and server-side energy consumption, the tool also sources its information from Recount. Preliminary tests show that frameworks such as ASP.NET Core and Ruby on Rails spawn multiple processes instead of just relying on threads. As energy measurements for a parent process do not include that of its subprocesses, the tool must first retrieve the Process Identifiers (PIDs) of its subprocesses before querying for their individual consumption data, and finally sum the results. The process-measuring tool is further detailed in Appendix B and its implementation can be found on GitHub [61].

1) Database Configuration: As mentioned in Section III, full-stack frameworks often provide their own ORM implementation, which facilitates type-safe and efficient interactions with an external database system. Because these frameworks do not ship their own database systems, testing their ORMs requires a separate database. PostgreSQL is one of the most widely used Database Management Systems (DBMSs), especially in professional environments [62]. Since PostgreSQL is supported by all of the ORMs tested in these frameworks, it is used as the database system for the benchmark.

Database systems such as PostgreSQL apply various optimization and caching strategies. To ensure that database interactions performed by one framework do not influence those of another framework, the database must be reset between these. The exact timing of database resets is illustrated in Algorithm 1, which depicts the different steps of the benchmark loop.

Both the benchmark runner and the process-measuring tool have been prepared for measuring a database instance. However, managing a PostgreSQL process while making sure that only the energy consumption related to the benchmark interactions is measured is out of the scope of this study. As the frameworks still need to interact with a database, a containerized Docker version of PostgreSQL is used. Using a containerized version removes the responsibility of managing a database from the benchmark runner, as its process is managed by Docker instead. This also simplifies database management, as it requires only a couple of CLI commands to start, stop, and reset it. When resetting the database, the container is stopped, after which both the container and associated volumes are deleted. Finally, its Compose script is run to start a fresh instance of the database. The database Compose script and data seed can be found in its GitHub repository [63].

D. Testing Environment

Apple Recount is used to measure both client-side and server-side energy consumption because it provides per-process measurements. As Recount is part of the X is Not Unix (XNU) kernel in macOS, the Device Under Test (DUT) must be an Apple-silicon-based Mac running macOS. The specifications of the DUT used for the benchmark are listed in Table IV.

Model	MacBook Air (2020)
Processor	M1 w. 8-core CPU, 7-core GPU, and 16-core Neural Engine
Memory	8 GB

TABLE IV: Hardware specifications of the DUT.

A clean install of macOS is utilized to ensure a stable test environment. During the benchmark, the DUT is permanently plugged in and constantly at a stable charge level. Additionally, to avoid inconsistencies in the energy consumption of other system components, the following system settings are used:

- Set *Display turn off Never* (Disables sleeping)
- Set *Start Screen Saver* to *Never*
- Disabled *Automatic adjust brightness*
- Disabled *True Tone*

The benchmarking process involves a variety of software tools. The utilized versions of these tools are, for the purpose of replicability, listed in Table V. Below each runtime, the

versions of frameworks using that runtime are displayed. The remaining dependencies of the different implementations are locked using their respective package managers and can be found in their respective repositories.

Name	Version
OS	macOS 26.4.1 Tahoe
Firefox	150.0.1
geckodriver	0.36.0
.NET	10.0.107
OpenJDK	25.0.3+9-LTS
Spring Boot	4.0.5
Python	3.14.4
Django	6.0.4
Node	24.11.1
Next	16.2.2
Astro	6.1.1
Ruby	3.4.9
Rails	8.1.3
turbo-rails	2.0.23
stimulus-rails	1.3.4
Docker	29.4.0
Postgres	17.9.1.pgdg13+1

TABLE V: Versions of software utilized during the test.

A stable browser testing environment is achieved by configuring Selenium to start Firefox with specific caching rules:

- `browser.cache.disk.enable` - `false`
- `browser.cache.memory.enable` - `true`
- `network.http.use-cache` - `true`

These rules allow caching within individual browsing sessions while disabling caching across browser restarts. This ensures that cached information from one test case cannot influence subsequent test cases.

To facilitate replication of the benchmark, a replication package is provided with step-by-step instructions [64].

E. Test Order

As described in Appendix B.B.5, each iteration uses the same order of frameworks and test cases. The specific order of test sites can be seen in Table VI. When software is tested repeatedly, changing system states can affect its energy consumption. If the same test order is used every iteration, such changes might introduce systematic biases. To mitigate this, a rotated round-robin order can be used for the test sites across iterations, allowing them to benefit equally from any order-dependent effects [65]. In [7], this is discussed as a threat to validity, but by inspecting the measurements, no evidence of systematic bias is found. Due to the similarity between the testing approach used in [7] and the one used in this study, we do not deem it necessary to use a rotated round-robin order.

Order	Test Site
1	ASP.NET HTMX
2	Ruby Rails Hotwire
3	Ruby Rails HTMX
4	Astro HTMX
5	Next.js
6	Django HTMX
7	Spring Boot HTMX

TABLE VI: Order in which the test sites are tested.

1) Warm-up Rounds: As part of the preliminary tests, the impact of just-in-time (JIT) compilation on the server-side energy consumption is explored. This is done because some runtimes may choose to optimize repeatedly executed code paths to improve performance. This is especially relevant for runtimes such as .NET's Common Language Runtime (CLR) and Java's Java Virtual Machine (JVM), whose execution models rely on JIT compilation for improving code performance [66].

As mentioned in Section III, full-stack frameworks are commonly used for large websites where they handle large amounts of requests. In this setting, the server is rarely restarted and therefore able to apply JIT-compilation for optimizing code paths related to frequently requested endpoints. Therefore, to ensure the test results are representative of a majority of the served page requests, each request path must be called multiple times to allow the JIT-compiler to optimize the code.

Figures 9 to 14 show a preliminary test performed using a single iteration with 31 rounds per test case. All of the charts show that a majority of the frameworks consume more energy in the initial round compared to subsequent rounds. In the case of ASP.NET and Spring Boot, this happens in multiple rounds before their consumption normalizes. As ASP.NET Core and Spring Boot stand out as taking more rounds before normalizing than the remaining frameworks, additional testing is performed on these.

To determine the number of rounds needed, a test of 45 iterations with 20 warm-up rounds is carried out using the ASP.NET and Spring Boot test sites. The results in Figures 15 to 20 show the average consumption for each round. Based on these results, we run a Mann-Whitney U test with binary segmentation, which tests every possible split point to find the most significant drop in p-value. This drop signifies that the framework has warmed up. To avoid Type I errors, we apply a

Bonferroni correction. The data for all test cases can be found in Appendix D. Based on the statistical analysis, the warm-up rounds are determined for each test case. The number of warm-up rounds is shown in Table VII.

Test-case	Warm-up rounds
static	11
live	9
list	3
list-interact	6
home-interact	7
navigate	8

TABLE VII: Warm-up rounds per test-case.

Every test case is performed for the specified number of warm-up rounds before the measured round is performed. By doing this, cold-start measurements are prevented from dominating the frameworks' measurements. The measured round is visualized in Figures 15 to 20 using vertical dotted lines.

VI. Experiments & Analysis

A. Data Analysis

The preliminary test consists of 60 iterations and ran for ~120 hours. The results of Cochran's formula, with a significance level of 0.05, and a standard deviation of $0.0570J$, show that 32 iterations are needed, meaning that the preliminary test is sufficient for further data analysis. Performing a Kolmogorov-Smirnov test on the results rejects the null-hypothesis of it being normally distributed with a p-value below $2.2e{-}16$. This is the expected result, and validates our use of non-parametric tests like Mann-Whitney U. Running the Kruskal-Wallis test as part of the Mann-Whitney U test shows that every test case has pairs of frameworks that use significantly different amounts of energy.

The average measurements from each test are shown in Tables VIII to X. Some statistics from the test cases are shown in Table XIII. An overview of which measurements are statistically significantly different is shown in Tables XI and XII.

Client-side energy consumption is significantly higher than server-side energy consumption; most of the client-side measurements are orders of magnitude larger than the server-side. The relative differences between test site measurements on the client side are relatively small, deviating up to 82 %. The relatively similar results also mean that there are few statistically significant differences, going as low as 12 out of 42 in the `list` test case. On the server side, the difference is relatively large, with the smallest difference being 394 % and the largest being 2856 %. There are also relatively many comparisons that show statistically significant differences. The test case with the fewest differences showed statistically significant differences in 36 out of 42 comparisons.

There is no clear ordering of test site performance. Generally speaking, the frameworks trade places depending on which resource type is measured and depending on the test case. Despite this, some generalizations can be made.

`astro-htmx` is the only test site that performs above-average in all test results, excluding those where statistical significance cannot be proven. It is the most efficient site in all 6 bandwidth tests.

`ruby-rails-hotwire` and `ruby-rails-htmx` are not as reliably efficient, but are arguably the next best performers, compared to the remaining options. In 5 of 6 test cases, they rank the best and second best on the server side, and in the last test case, they rank second and third best behind `astro-htmx`. Considering client-side consumption, `ruby-rails-hotwire` has mixed performance while `ruby-rails-htmx` performs in the range from average to below-average. Considering bandwidth consumption, `ruby-rails-htmx` performs average, and `ruby-rails-hotwire` performs below-average. There is no single test case metric in which they are the worst performers.

`nextjs` is the only test site that performs below-average across all test case metrics (except one). It is the least bandwidth-efficient site in all 6 test cases, the least energy-efficient on the client side in 5 out of 6 test cases, and the least energy-efficient on the server side in 2 out of 6 test cases. It is, however, the next-best performer in client-side energy consumption for the `list-interact` test case.

We make no attempt to generalize `spring-boot-htmx`, `django-htmx`, and `asp-net-htmx` since they all have a mix of above-average and below-average results. We can, however, conclude that each framework is the worst performer in at least one test case.

The data is analyzed in further detail in Section VIII.

Framework	home-interact	list-interact	list	live	navigate	static
ruby-rails-hotwire	0.9066	10.0782	0.0820	0.2334	0.4039	0.1068
asp-net-htmx	0.9114	11.1932	0.0780	0.2214	0.4173	0.0997
astro-htmx	0.9525	11.1644	0.0895	0.2249	0.4030	0.0994
spring-boot-htmx	0.9864	11.2516	0.0800	0.2218	0.4110	0.0971
ruby-rails-htmx	1.0028	11.1395	0.0876	0.2233	0.4201	0.0997
django-htmx	0.9994	11.0988	0.0910	0.2270	0.4172	0.1013
nextjs	1.2292	10.6412	0.1419	0.3179	0.5460	0.1603

TABLE VIII: Average client-side energy consumption (J) for all frameworks across all test cases.

Framework	home-interact	list-interact	list	live	navigate	static
ruby-rails-htmx	0.0127	0.0142	0.0031	0.0134	0.0435	0.0037
ruby-rails-hotwire	0.0183	0.0178	0.0045	0.0120	0.0428	0.0044
astro-htmx	0.0490	0.0660	0.0069	0.0078	0.0646	0.0067
asp-net-htmx	0.1497	0.0975	0.0262	0.0366	0.2112	0.0438
django-htmx	0.1729	0.2292	0.0211	0.0336	0.0789	0.0175
nextjs	0.1239	0.1649	0.0462	0.0870	0.1003	0.0481
spring-boot-htmx	0.3741	0.2251	0.0859	0.0644	0.1870	0.0351

TABLE IX: Average server-side energy consumption (J) for all frameworks across all test cases.

Framework	home-interact	list-interact	list	live	navigate	static
astro-htmx	44.3	59.6	26.1	25.1	48.9	27.5
django-htmx	52.6	66.3	36.2	35.1	58.2	36.6
spring-boot-htmx	61.9	66.5	34.2	44.8	64.6	35.7
ruby-rails-htmx	71.1	78.3	45.5	44.0	70.2	45.7
asp-net-htmx	74.3	73.0	40.8	58.2	79.4	42.4
ruby-rails-hotwire	78.9	91.6	57.5	56.1	80.3	57.7
nextjs	312.5	290.1	181.0	194.5	224.2	197.1

TABLE X: Average client-side bandwidth consumption (kilobyte (kB)) for all frameworks across all test cases.

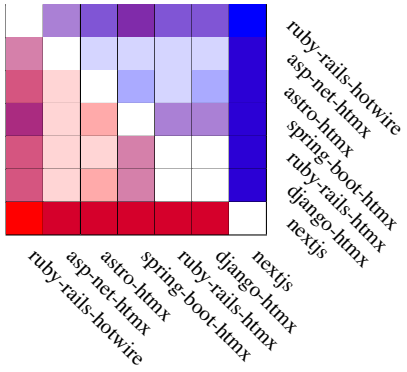


TABLE XI: Comparison matrix for client-side test site consumption for all benchmarks. *Color legend:* Red = row has statistically significant higher energy consumption than column in all tests; Blue = row has statistically significant lower energy consumption than column in all tests; White = row and column has no statistically significant difference in all tests.

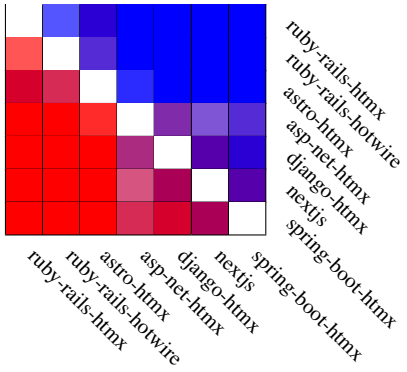


TABLE XII: Comparison matrix for server-side test site consumption for all benchmarks. *Color legend:* Red = row has statistically significant higher energy consumption than column in all tests; Blue = row has statistically significant lower energy consumption than column in all tests; White = row and column has no statistically significant difference in all tests.

Metric	home-interact	list-interact	list	live	navigate	static
Client energy - Max difference (%)	36	12	82	44	35	65
Client energy - Average standard deviation (J)	0.0580	0.1179	0.0290	0.0342	0.0391	0.0165
Client energy - Statistically Significant diffs (of 42)	38	28	12	28	22	14
Server energy - Max difference (%)	2856	1513	2656	1021	394	1183
Server energy - Average standard deviation (J)	0.0322	0.0145	0.0136	0.0132	0.0285	0.0098
Server energy - Statistically significant diffs (of 42)	38	42	40	40	40	36
Bandwidth - Max difference (%)	606	387	593	674	358	617
Bandwidth - Average standard deviation (kB)	0.0274	0.0236	0.3432	0.0129	0.0406	0.0128

TABLE XIII: Statistics for test cases.

VII. Threats to Validity

A. Database Energy Consumption

As mentioned in Section V-C, both client and server processes are measured during the benchmark. This does, however, not include the database, as it runs as a separate process, which is not controlled by the server process. There are multiple ways in which the different test sites might differ in energy consumption:

- Different frameworks' ORM might create different queries for obtaining the same data, leading to different amounts of energy consumed by the database.
- Different frameworks' ORM might not exclusively rely on the database for manipulating its contents. Instead, some operations may be performed in the application layer instead of solely in the database layer.

B. WebDriver Interactions

As mentioned in Section V-B, Selenium is used for automating interactions with the Firefox browser. Selenium translates calls made using its API to the equivalent WebDriver requests and sends them to the provided geckodriver instance. The WebDriver protocol is a W3C standard that enables programmatic interactions with supported browsers. Upon receiving a WebDriver request, the GeckoDriver converts it into the Marionette protocol, which is a protocol used for automating Gecko instances, and sends them to the controlled Firefox instance.

When a user interacts with an instance of Firefox, those interactions are delivered as OS-level events. However, when Firefox receives Marionette requests, they are directly executed in its Gecko engine [67]. This means that the two types of interactions might not be handled the same internally, leading to differences in the per-interaction energy consumption. This, however, does not affect the energy consumption comparison, as all of the frameworks are tested using the same automated interaction sequence.

C. Recount Accuracy

As mentioned in Section V-C, the Recount subsystem from the XNU kernel is utilized to source process-isolated energy measurements for both the client and the server. Intel RAPL-based alternatives to the Recount subsystem are discussed in Appendix B.A.1. These energy attribution methods were ultimately not used due to the risk of introducing estimation inaccuracies, which are not present when using Recount. Additionally, using Recount for both the client and server makes their measurements directly comparable.

A limitation of using Recount over RAPL is that its absolute energy-consumption accuracy has not yet been tested in a study, restricting comparisons to relative comparisons between same-device measurements. As mentioned in Section V-D, we use the same device to obtain all measurements, and environment adjustments have been made to ensure that it is consistent across benchmark iterations. As the measurements are only used for relative consumption comparisons, this restriction does not affect the results of this study.

D. Testing a Single Browser

A consequence of using the Gecko Profiler for measuring client-side energy consumption is that browser support is limited to Gecko-based browsers. As a result, we are unable to cross-examine other browsers to determine whether client-side energy consumption differs from the measurements obtained in Firefox. This limitation restricts our ability to generalize the client-side energy consumption observations to different browsers and runtime environments.

E. Web Development Experience

It is central to the benchmark that the test sites are implemented to the same standard. As we are experienced in

both ASP.NET, Astro, and Next.js, but not in Ruby, we risk implementing the known frameworks more efficiently. We must therefore be careful not to introduce an unfair bias that may affect their performance.

To prevent potential configuration bias, the test sites are created using the framework's official scaffolding tools. This ensures that the test sites use configurations that are recommended by the frameworks.

To prevent potential bias as a result of prior framework experience, development guidelines are defined in Appendix A.B. These guidelines ensure that a similar development approach is used for all test sites, that implementations follow the framework's recommended practices, and that another developer reviews the implementations before they are tested. To help enforce the guidelines discussed in Appendix A.B, a site-verifier is developed. This tool automatically interacts with the test sites to ensure that they use the expected request headers, are structured like the reference site, and act as expected upon list interactions. The tool is further detailed in Appendix A.B.2.

Before the individual test sites are marked as ready, preliminary tests are performed and analyzed for unexpected results. If unexpected results are discovered, they are further investigated and fixed if necessary. This process is repeated until we no longer discover issues.

By applying these mitigation strategies, the risk of introducing bias based on prior experience is minimized.

F. Test Site Design

The test site design is based on human observation and interpretation, which could be biased. To limit this, we followed a strict process of categorization described in Section IV, to ensure each website was represented and that we documented each decision, making it possible to double-check the work and detect possible shortcomings.

G. Test Case Representativeness

When running the test cases, one approach is to perform interactions immediately one after another. Many front-end frameworks will batch successive user interactions together to skip unnecessary work [68, 69]. They will also speculatively prepare for certain user interactions based on the position of the cursor. To account for this, delays are added between each interaction so that they are not batched, enabling the use of speculative preloading.

H. Test Order

As discussed in Section V-E, the test sites do not follow a rotated round-robin order across the iterations. To check whether the energy measurements contain systematic biases, the results are inspected. Appendix F contains a chart for every test case, in which linear regressions for each test site are visualized. While a majority of these charts do not suggest any trends, all of the regressions performed on the `static` test case suggest a decrease in energy consumption as iteration count increases. Additionally, the coefficients of determination (R^2) in this test case are closer to 1 compared to the remaining test cases.

As the `static` test case represents the minimum workload for the client and server, its energy consumption is correspondingly low, making it especially sensitive to external influence. By plotting all test site results for this test case, it is observed that the data contains a downward trend in energy consumption. The clearest example of such a trend can be seen in Figure 1, which visualizes the measurements of *django-htmx*, one of the test sites with the highest coefficients of determination (R^2).

Metric (J)	Experiment 1		Experiment 2	
	1.Django	2.Hotwire	1.Hotwire	2.Django
Combined AVG	0.1416	0.1108	0.1349	0.1176
Combined SD	0.0144	0.0120	0.0113	0.0107
Client AVG	0.1181	0.1063	0.1293	0.0996
Client SD	0.0140	0.0114	0.0113	0.0084
Server AVG	0.0235	0.0045	0.0055	0.0180
Server SD	0.0012	0.0011	0.0008	0.0029

TABLE XIV: Energy consumption results from performing two experiments in which the static test case was run 500 times on the *django-htmx* and *ruby-rails-hotwire* test sites. The first experiment initially performed 500 iterations of `static` on *django-htmx* and then performed 500 iterations on *ruby-rails-hotwire*. The second experiment reversed this order. All measurements are reported in joules (Js).

To investigate this trend further, we conduct an experiment with two test sites running back-to-back. For each test site, 500 consecutive iterations of the `static` test case are performed. The experiment is carried out twice, with the test site order reversed in the second run. We use `django-htmx` and `ruby-rails-hotwire` due to their high coefficients of

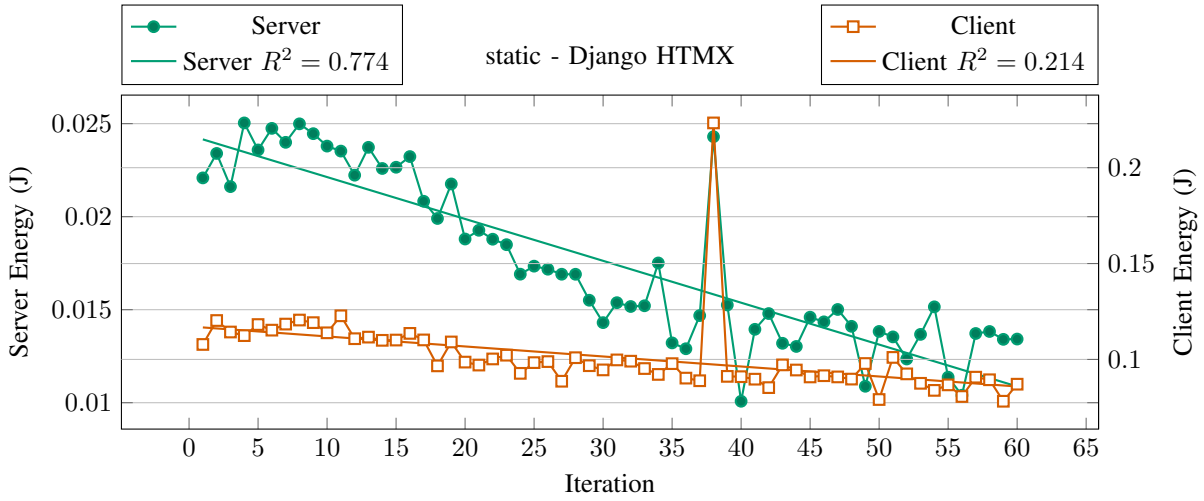


Fig. 1: Client-side and server-side energy consumption over 60 iterations of the static test case. Data sourced from the benchmark results. Linear regressions were performed to make it easier to detect trends.

determination (R^2) in the `static` test case. The results are listed in Table XIV.

Inspecting the results reveals that the latter framework benefits from the test order and consequently that the system state evolves over time to become more efficient. By interleaving the test sites in the benchmark loop, rather than performing all iterations on one test site before moving on to the next, the systematic bias resulting from the gradually changing system state has been minimized. Future benchmarks should adopt a rotated round-robin order to further mitigate the risk of systematic bias.

Another notable observation from Figure 1 is the sudden peak in iteration 38, in which both the server-side and client-side energy consumption peak. Although outlier detection has not been applied, the statistical significance analysis remains valid due to the use of the Mann-Whitney U test, which is not very sensitive to such peaks. The test is non-parametric and thus uses rankings for each of the measurements. This means that we compare the median of the results from each test site and not the mean, thus minimizing the impact of the outliers. These peaks might, however, still influence the remaining comparisons, which are based on the raw results, meaning that not handling outliers remains a limitation.

I. JIT Compiled Languages

Languages such as C# and Java are JIT-compiled, which means the runtime compiles the source code as it is being executed. Each execution of the code can vary significantly in energy consumption because the measurement, in some cases, includes the compilation of the source code. This also means that the compiled source code can change between executions. To emulate a typical execution in a production environment, we implement warm-up rounds that allow the compiler to reach a steady state where the code is already compiled and cached, before measuring the energy usage of the execution.

VIII. Discussion

A. Patterns in Full-Stack Energy Consumption

The results show that statistically significant differences are present in all test cases for both client-side and server-side comparisons. The differences are especially frequent in server-side comparisons and less frequent in client-side comparisons. This answers RQ 1 about whether there is a statistically significant difference in the energy consumption of websites built with different full-stack frameworks. Answering RQ 2, the results presented in Section VI, show that frameworks perform differently and that there is an overall trend in which frameworks consume less energy.

Comparing the client-side energy consumption shows few statistically significant differences in most of the test cases. Most of the differences are seen when comparing test sites with different front-end runtimes. That is reasonable, given that the test sites with the same HTMX front-end should be similar to each other in performance. `nextjs` performs poorly in most test cases, but is more efficient on the client-side in the `list-interact` test case when compared to all HTMX-based sites. This indicates that the React runtime has a larger baseline cost, but that the cost can be offset because it is

more efficient in cases where it needs to replace large amounts of data in HTML structures. `ruby-rails-hotwire` performs better than the other test sites in the more advanced `list-interact` and `home-interact` test cases, but otherwise performs worse or identically compared to other test sites in the less advanced test cases. This finding is similar to that of `nextjs`, but the better overall results indicate that Hotwire has a lower baseline cost.

Comparing the server-side energy consumption shows a larger difference between the test sites. Most test sites are ranked the same across all test cases, but a few stand out. `django-htmx` performs average in most, but performs worst or second worst in `list-interact` and `home-interact`. This indicates that it is inefficient at querying the database. `nextjs` performs slightly better in the `list-interact` `home-interact`, and `navigate` test cases. This indicates that it is relatively inefficient at serving the first page request, but relatively efficient at swapping page content. This is as expected, given that Next.js uses a larger runtime with a high initial cost, which can efficiently stream differences between two pages and update the HTML structure on the page to match the new page.

Comparing the bandwidth consumption, shown in Table X, reveals that there are more test cases in which the test site ordering is the same, compared to the energy consumption results. This indicates that the website design and usage pattern do not factor into the bandwidth efficiency, allowing us to draw more general conclusions on this topic. Five of these test sites use almost identical setups with the same HTML structure and an HTMX runtime for client-side interactivity. Despite this, `ruby-rails-htmx` uses almost double the bandwidth of `astro-htmx`. To understand this discrepancy, Table XV has been created to show an overview of the size of the client-side runtimes used in each test site.

The first column in Table XV shows the plaintext size of the runtime shipped to the client. For `django-htmx`, `spring-boot-htmx`, and `asp-net-htmx`, the HTMX source code must be downloaded manually, and HTMX recommends using the version that is minified by HTMX. This means all three pages use the exact same runtime at a little over 95 kB. The small differences in size are caused by minor extensions that are manually written for HTMX to make it work with each framework. In `astro-htmx`, the recommended method is to install HTMX via npm, which uses the unminified source code, but then Astro performs its own minification before pushing the runtime to the client. Astro's minification is superior to the minification used by the HTMX developers. This can partly be explained by the fact that Astro can also minify the extensions that were manually written, which is not possible in the other implementations. In `ruby-rails-htmx`, it is also recommended to use framework-specific commands to fetch the source code from npm. But the recommended default in Ruby on Rails is to use the unminified source code directly, resulting in the large code size [70].

The second column in Table XV shows the transfer size of the runtime. This is the actual amount of data sent over the internet after the framework has compressed the code. After transmission, the client side must decompress the data to read the source code. Each framework has its own setup for compression, but as seen in column four, most rely on the gzip compression standard, while `astro-htmx` uses the newer brotli compression standard. The third column shows that each framework is able to compress the runtime into between a third and a fourth of the original size. The differences in minification strategies explain how there can be such a big difference between otherwise identical sites.

In `ruby-rails-hotwire`, HTMX is replaced with Hotwire. The Hotwire runtime is slightly larger, resulting in higher bandwidth consumption. In `nextjs`, it uses a similar bundling and minification strategy to `astro-htmx`, but the runtime is almost ten times larger, resulting in higher bandwidth consumption. The small size of HTMX, combined with the aggressive minification system in Astro, results in `astro-htmx` being the most bandwidth-efficient test site.

B. Exploratory Combined Energy Usage Analysis

The results show that different frameworks perform the best depending on which metrics are prioritized. One way to prioritize is to combine all three metrics into a combined energy consumption measurement, allowing metrics to be prioritized based on their scale relative to other metrics.

To make this combined measurement, the energy consumption of transferring the bandwidth over the global internet infrastructure has to be estimated. According to [71], the energy consumption between server/client vs the network is split 76 % for the server/client and 24 % for the network. By knowing the total energy consumption across all test site results and test case metrics on the client and server, it is possible to extrapolate the total energy consumption on the network:

Test site	Runtime size (kB)	Transfer size (kB)	Compression ratio	Compression standard
astro-htmx	65.33	20.82	3.14	brotli
django-htmx	95.41	25.21	3.78	gzip
spring-boot-htmx	95.67	25.42	3.76	gzip
asp-net-htmx	95.87	30.71	3.12	gzip
ruby-rails-htmx	128.11	32.91	3.89	gzip
ruby-rails-hotwire	155.94	47.26	3.30	gzip
nextjs	558.78	165.03	3.39	gzip

TABLE XV: Overview of compression and minification methods employed on the client-side runtime on the static subpage.

$92.7093J/76*24 = 29.2766J$. Because we also know the total bandwidth consumption across all tests, we can calculate how many joules are used per kB: $29.2766J/3368.6kB = 0.08691$. This allows us to convert kB to joules and sum all three metrics.

The client-side has the highest energy consumption, meaning it naturally has a bigger influence on the results. The bandwidth consumption is not as big, but still big enough to influence the results. Lastly, given the relatively small energy consumption on the server side, it has the least influence on the results. Because most of the client-side energy measurements are so similar to each other, we expect the bandwidth results to have the largest effect on the final rankings of each test site.

The result of the final calculation is shown in Table XVI. As expected, the final ranking of each test site matches the ranking of test sites when measuring bandwidth consumption, with the only exception being that `ruby-rails-htmx` and `ruby-rails-hotwire` are slightly more efficient than `spring-boot-htmx` and `asp-net-htmx`. These test sites are close enough in client-side and bandwidth consumption that the larger difference in server-side consumption is enough to change their ranking. The conclusions from this list are largely unaltered from the conclusions made in Section VI; `astro-htmx` is still the most efficient and `nextjs` is still the least efficient. The only discrepancy is that `django-htmx` is the next most efficient test site, being slightly more efficient compared to `ruby-rails-htmx` and `ruby-rails-hotwire`. `django-htmx` was previously not considered to be among the best-performing frameworks because of its average and unstable energy consumption on both the client-side and server-side across all test cases. But because it is the next most efficient in bandwidth measurements, and because those results have an outsized effect on these rankings, it is now the next most efficient test site in Table XVI.

C. Comparing Results to Recent Literature

As mentioned in Section II, previous research has focused more on the client-side. In [7], front-end frameworks are measured using a similar methodology. The results of this benchmark showed that the Astro framework was one of the best-performing out of the 18 front-end framework configurations benchmarked, and Next.js was the overall worst. When compared to this paper, Astro is also one of the best performing, with Next.js being the worst. In [6], they test the frameworks Qwick and Next.js, also using the Gecko Profiler [72]. The results showed that Qwik used 6-10 % less energy than Next.js, aligning with our findings on Next.js' consumption relative to the other Frameworks. Finally, the React runtime is deemed the worst-performing out of the JavaScript frameworks tested in [8], where they measure the rendering performance of frameworks.

D. Our Recommendations on Framework Choice

We acknowledge that choosing a framework is complex and has multiple factors like maturity, documentation, and support [73]. The recommendations we provide are based on energy consumption alone. Based on the results, we recommend choosing Astro as it is the overall least energy-consuming framework. The tests show that the client-side consumption is larger than the server-side consumption, but if the server-side consumption is deemed more important, we recommend using Ruby on Rails with HTMX. If choosing a full-stack framework is a must, and thus removing Astro as a choice, we recommend either Ruby on Rails or Django.

E. Potential for improvement in the ICT sector

Table XVIII gives an impression of how often frameworks are used. 81 % of the frameworks listed are tested in this paper. Out of those, 6 % use Next.js, which is the most inefficient framework, and 73 % of them use Spring or ASP.NET Core, which are not the most efficient frameworks. These findings

Test site	home-interact	list-interact	list	live	navigate	static
astro-htmx	1.3865	11.7484	0.3232	0.4508	0.8926	0.3451
django-htmx	1.6294	11.9042	0.4267	0.5657	1.0019	0.4369
ruby-rails-htmx	1.6334	11.8342	0.4861	0.6191	1.0737	0.5006
ruby-rails-hotwire	1.6106	10.8921	0.5862	0.7330	1.1446	0.6127
spring-boot-htmx	1.8985	12.0547	0.4631	0.6756	1.1594	0.4425
asp-net-htmx	1.7068	11.9251	0.4588	0.7638	1.3186	0.5120
nextjs	4.0690	13.3274	1.7612	2.0953	2.5948	1.9214

TABLE XVI: Average estimated combined energy consumption (J) for all test sites across all test cases.

are corroborated by [24], which shows that developers work significantly more in the inefficient frameworks than in the efficient frameworks. Combined, they show that there is a large potential for reducing the resource consumption of the ICT sector by switching to more efficient full-stack frameworks.

F. Future Work

1) DBMS Energy Consumption: Section VII-A describes the threat to validity of not measuring the energy consumption of the DBMS. To mitigate this threat, future studies could apply the process-measuring tool for the DBMS to measure its energy consumption during the benchmark. This could additionally provide further insight into the efficiency of the different ORMs.

2) ORM Energy Consumption: As mentioned in Section III-C, not all frameworks ship with their own ORM, resulting in us having to choose one. Drizzle is ultimately chosen over alternatives based on its popularity, usage, and performance. To enable making energy conscious ORM decisions, future studies could explore the energy efficiency of different ORMs.

3) Varying the Test Platform: The process-measuring tool is currently limited to supporting Apple Silicon-powered Macs running macOS. Future studies could expand its support to other platforms by implementing alternative energy measurement sources. For example, by utilizing a combination of Intel RAPL and one of the energy attribution models discussed in Appendix B.A.1, support can be expanded to Intel-based platforms running Linux. Evaluating energy consumption on other platforms could provide valuable insights into the platform-dependency of the different frameworks' relative energy consumption.

4) Varying the Tested Browser: As discussed in Section VII-D, the client-side energy consumption findings are limited to the Firefox browser. To evaluate the generalizability of the client-side findings, future studies could expand testing to additional browsers.

5) Explore Effects of JIT Compilation: As discussed in Section V-E1, the runtimes are allowed to apply JIT compilation before measuring their energy consumption. This is facilitated by running a fixed number of warm-up rounds before measuring their energy consumption in the subsequent round. To ensure that this approach accurately captures the long-term behavior of the JIT-compiled runtimes, future studies could explore the consequences of extending the number of measured rounds. Such studies could also assess the feasibility and accuracy of measuring multiple rounds after the warm-up rounds, which could decrease the total benchmarking time.

IX. Conclusion

This paper contributes to state-of-the-art energy efficiency research by comparing the energy efficiency of seven popular full-stack framework configurations using a benchmark runner and a novel test site to carry out benchmarks in a replicable manner. To achieve this, we answer the following RQs:

RQ 1 Is there a statistically significant difference in the client-side and server-side energy consumption of web applications built with different full-stack frameworks?

RQ 2 Are there any observable patterns in the energy consumption difference?

With supporting RQs:

RQ 3 How can the energy consumption of both the front-end and back-end of a website be measured?

RQ 4 How can a test site be designed and implemented to be representative of the most visited full-stack framework-based websites?

RQ 5 How can test cases be designed to include web components commonly found on the most visited websites built using full-stack frameworks?

A. Automation Tools

To automate the benchmarking process whilst measuring both client-side and server-side energy consumption, the benchmark runner from [7] is extended. It uses Selenium to interact with the test site in Firefox, and the Gecko Profiler to gather per-tab energy consumption measurements on the Apple Silicon-powered Mac. To answer RQ 3, measuring the back-end is achieved using a custom process-measuring tool, which sources its energy measurements from the same subsystem as the Gecko Profiler, i.e., Apple Recount. The benchmark runner is further extended to maintain a database for the full-stack frameworks, and to support warm-up rounds that enable JIT-compiled frameworks to apply their optimizations.

B. Test Sites

To determine which frameworks to test, the 100 most visited websites are inspected, filtered by whether or not they use a full-stack framework, and categorized by content type. Six frameworks are chosen; Astro, Ruby on Rails, Django, Next.js, ASP.NET Core, and Spring Boot. These are chosen based on their usage in the 100 most visited sites and their popularity among developers.

To answer RQ 4, we inspect the identified categories and their common web elements. A five-subpage test site that represents all of these categories is designed and implemented for each configuration. To ensure the test sites are equivalently designed, development guidelines are created, and an automated site-verifier is developed and used.

The test sites are evaluated with six test cases, each of which tests the frameworks using different workloads. By designing the test cases to interact with the elements commonly found in each of the categories, the test cases are able to evaluate their energy consumption, answering RQ 5.

C. Results

Analyzing the benchmark results reveals statistically significant differences in the client-side and server-side energy consumption between full-stack frameworks, answering RQ 1. The largest difference is observed on the server side, where the frameworks differ up to 2856 % and a majority of test case comparisons are statistically significantly different. In contrast, it is often not possible to prove a statistically significant difference between the test cases on the client-side, with as few as 12 out of 42 comparisons being different in one of them. This is a result of using the same front-end framework across almost all configurations to facilitate direct comparisons, which also leads to the energy consumption only deviating by up to 82 %. These findings allow taking energy consumption into account when choosing between full-stack frameworks.

Overall, Astro and Ruby on Rails perform the best, while Next.js performs the worst. Astro stands out because of its above-average performance across all tests, and its very low bandwidth usage. Next.js performs especially poorly on the client side and has high bandwidth consumption due to its large React runtime. By estimating the energy consumption of the frameworks' bandwidth usage, it is possible to evaluate their combined consumption. From their combined energy usage, the frameworks Astro and Django are found to consume the least amount of energy. This answers RQ 2.

An analysis of the most used and most popular frameworks reveals that some of the most inefficient full-stack frameworks are in widespread use, and thus there is a large potential for improving the energy efficiency in the ICT sector by switching to more efficient full-stack frameworks.

BIBLIOGRAPHY

[1] Markus Dick, Stefan Naumann, and Alexandra Held. "Green Web Engineering - A Set of Principles to Support the Development and Operation of "Green" Websites and their Utilization during a Website's Life Cycle." In: *WEBIST 2010 - Proceedings of the 6th International Conference on Web Information Systems and Technology*. Vol. 1. 2010, pp. 48–55. URL: https://www.researchgate.net/publication/220724448_Green_Web_Engineering_-_A_Set_of_Principles_to_Support_the_Development_and_Operation_of_Green_Websites_and_their_Utilization_during_a_Website's_Life_Cycle.

[2] Robert Istrate et al. "The environmental sustainability of digital content consumption". In: *Nature Communications* 15.1 (2024), p. 3724. DOI: 10.1038/s41467-024-47621-w.

[3] Kwame Chan-Jong-Chu et al. "Investigating the Correlation between Performance Scores and Energy Consumption of Mobile Web Apps". In: *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2020, pp. 190–199. DOI: 10.1145/3383219.3383239.

[4] Mohammad Ghattas, Suhail Odeh, and Antonio M. Mora. "Predicting Website Performance: A Systematic Review of Metrics, Methods, and Research Gaps (2010–2024)". In: *Computers* 14.10 (2025). DOI: 10.3390/computers14100446.

[5] Statista. *Most used web frameworks among developers 2025| Statista*. URL: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/ (visited on 04/22/2026).

[6] Janne Kalliola and Juho Vepsäläinen. "Challenges Related to Approximating the Energy Consumption of a Website". In: *IEEE Access* 13 (2025), pp. 139001–139017. DOI: 10.1109/ACCESS.2025.3596459.

[7] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Comparing Energy Consumption of Front-End Web Frameworks*. 2026.

[8] Risto Ollila, Niko Mäkitalo, and Tommi Mikkonen. "Modern Web Frameworks: A Comparison of Rendering Performance". In: *Journal of Web Engineering* 21.3 (2022), pp. 789–813. DOI: 10.13052/jwe1540-9589.21311.

[9] Abdulaziz Aborujilah et al. "Descriptive Analysis of Built-in Security Features in Web Development Frameworks". In: *2022 16th International Conference on Ubiquitous Information Management and Communication (IMCOM)*. 2022. DOI: 10.1109/IMCOM53663.2022.9721750.

[10] Zakaria Mehrab et al. "Mining Developer Questions about Major Web Frameworks". In: *International Conference on Web Information Systems and Technologies*. 2018. DOI: 10.5220/0006929501910198.

[11] Imad Buljic et al. "Comparative Performance Analysis of Leading Backend Frameworks for Developers". In: *2025 24th International Symposium INFOTEH-JAHORINA (INFOTEH)*. 2025, pp. 1–5. DOI: 10.1109/INFOTEH64129.2025.10959250.

[12] Mateusz Szewczyk and Maria Skublewska-Paszkowska. "Performance comparison of development frameworks in selected environments in REST API architecture". In: *Journal of Computer Sciences Institute* 35 (2025), pp. 121–128. DOI: 10.35784/jcsi.7041.

[13] Ostap Kilbasovych et al. "DEMO: REST-Q -A Framework to Assess Energy Consumption in Web Applications". In: *Proceedings of the 2025 IEEE International Conference on Cloud Engineering IC2E 2025*. IEEE. 2025. DOI: 10.1109/IC2E65552.2025.00020.

[14] Sergio Di Meglio and Luigi Starace. "Evaluating Performance and Resource Consumption of REST Frameworks and Execution Environments: Insights and Guidelines for Developers and Companies". In: *IEEE Access* PP (2024), pp. 1–1. DOI: 10.1109/ACCESS.2024.3489892.

[15] Narendran Thiagarajan et al. "Who killed my battery: Analyzing mobile browser energy consumption". In: *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web* (2012). DOI: 10.1145/2187836.2187843.

[16] João de Macedo et al. "Energy Wars - Chrome vs. Firefox: Which browser is more energy efficient?" In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 2020, pp. 159–165. DOI: 10.1145/3417113.3423000.

[17] Matteo Varvello and Benjamin Livshits. *On the Battery Consumption of Mobile Browsers*. 2020. eprint: 2009.03740 (cs.OH). URL: https://arxiv.org/abs/2009.03740.

[18] Anthony DiBenedetto and Fadi Wedyan. "An Empirical Evaluation of Energy Consumption Across Web Browsers". In: *2025 10th International Conference on Fog and Mobile Edge Computing (FMEC)*. 2025, pp. 39–48. DOI: 10.1109/FMEC65595.2025.11119363.

[19] Louay Khrouf et al. “On the Energy Consumption of Web Applications: An Empirical Study of their Design Solutions”. In: *2025 IEEE/ACM 9th International Workshop on Green and Sustainable Software (GREENS)*. 2025, pp. 47–54. DOI: 10.1109/GREENS66463.2025.00012.

[20] Nicolas Sevcik. *Carbon Impact: An app for continuous estimation of carbon emissions of servers*. eng. 2024. URL: https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-72871.

[21] Benoit Petit. *Scaphandre*. URL: https://github.com/hubblo-org/scaphandre (visited on 02/04/2026).

[22] Hongyu Hè, Michal Friedman, and Theodoros Rekatsinas. “EnergAt: Fine-Grained Energy Attribution for Multi-Tenancy”. In: *Proceedings of the 2nd Workshop on Sustainable Computer Systems*. HotCarbon ’23. ACM, 2023, pp. 1–8. DOI: 10.1145/3604930.3605716.

[23] Benjamin Weigell, Simon Hornung, and Bernhard Bauer. *METRION: A Framework for Accurate Software Energy Measurement*. 2025. DOI: 10.48550/arXiv.2512.06806. arXiv: 2512.06806 [cs.SE].

[24] Stack Overflow. *Developer Survey*. Tech. rep. Stack Overflow, 2025. URL: https://survey.stackoverflow.co/2025/technology.

[25] MDN. *Server-side web frameworks*. URL: https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Web_frameworks.

[26] Django. *Django FAQ*. URL: https://docs.djangoproject.com/en/6.0/faq/general/.

[27] Next.js. *Next.js Docs*. URL: https://nextjs.org/docs.

[28] Microsoft. *Razor Pages architecture and concepts in ASP.NET Core*. URL: https://learn.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-10.0&tabs=visual-studio.

[29] Astro. *Why Astro?* URL: https://docs.astro.build/en/concepts/why-astro/.

[30] Vishnuvardhan Goli and Research V. “Enhancing React Development Server Components and Performance Best Practices”. In: *International Journal of Advanced Research in Engineering and Technology* 11 (2020), pp. 649–660. DOI: 10.34218/IJARET_11_02_071.

[31] Meta. *Server Components*. URL: https://react.dev/reference/rsc/server-components (visited on 05/27/2026).

[32] Hotwire. *Hotwire - HTML Over The Wire*. URL: https://hotwired.dev/ (visited on 05/27/2026).

[33] Stack Overflow. *Developer Survey*. Tech. rep. Stack Overflow, 2024. URL: https://survey.stackoverflow.co/2024/technology.

[34] HTMX. *HTMX - High Power Tools for HTML*. URL: https://htmx.org/ (visited on 05/27/2026).

[35] Azmain Adel. *Prisma v Drizzle Benchmark on a PostgreSQL DB*. 2026. URL: https://github.com/azmainadel/prisma-v-drizzle-pg-benchmark (visited on 05/21/2026).

[36] Drizzle. *Drizzle ORM - Benchmarks*. 2026. URL: https://orm.drizzle.team/benchmarks (visited on 05/21/2026).

[37] Drizzle. *Postgres benchmarks between Drizzle ORM and other popular ORM libraries*. 2026. URL: https://github.com/drizzle-team/drizzle-northwind-benchmarks-pg (visited on 05/21/2026).

[38] Prisma. *prisma - npm*. 2026. URL: https://www.npmjs.com/package/prisma (visited on 05/21/2026).

[39] Drizzle. *drizzle-orm - npm*. 2026. URL: https://www.npmjs.com/package/drizzle-orm (visited on 05/21/2026).

[40] Kysely. *kysely - npm*. 2026. URL: https://www.npmjs.com/package/kysely (visited on 05/21/2026).

[41] TypeORM. *typeorm - npm*. 2026. URL: https://www.npmjs.com/package/typeorm (visited on 05/21/2026).

[42] Knex.js. *knex - npm*. 2026. URL: https://www.npmjs.com/package/knex (visited on 05/21/2026).

[43] MikroORM. *mikro-orm - npm*. 2026. URL: https://www.npmjs.com/package/mikro-orm (visited on 05/21/2026).

[44] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Test Site - Ruby Rails Hotwire*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/test-site-ruby-rails-hotwire.

[45] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Test Site - Ruby Rails HTMX*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/test-site-ruby-rails-htmx.

[46] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Test Site - Astro HTMX*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/test-site-astro-htmx.

[47] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Test Site - ASP.NET HTMX*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/test-site-asp-net-htmx.

[48] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Test Site - Spring Boot HTMX*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/test-site-spring-boot-htmx.

[49] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Test Site - Django HTMX*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/test-site-django-htmx.

[50] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Test Site - Next.js*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/test-site-nextjs.

[51] Next.js. *Prefetching*. URL: https://nextjs.org/docs/app/guides/prefetching (visited on 06/05/2026).

[52] William G. Cochran. *Sampling Techniques*. 3rd. John Wiley & Sons, 1977. ISBN: 978-0-471-16240-7.

[53] James E. Barlett, Joe W. Kotrlik, and Chadwick C. Higgins. "Organizational research: Determining appropriate sample size in survey research". In: *Information Technology, Learning, and Performance Journal* 19.1 (2001), pp. 43–50. URL: https://www.proquest.com/scholarly-journals/organizational-research-determining-appropriate/docview/219816871/se-2.

[54] A. Kolmogorov. "Sulla determinazione empírica di una legge di distribuzione". In: *Giornale dellÍstituto italiano degli attuari* (1933). URL: http://digitale.bnc.roma.sbn.it/tecadigitale/giornale/CFI0353791/1933/unico/00000093.

[55] William H. Kruskal and W. Allen Wallis. "Use of Ranks in One-Criterion Variance Analysis". In: *Journal of the American Statistical Association* 47.260 (1952), pp. 583–621. DOI: 10.2307/2280779.

[56] H. B. Mann and D. R. Whitney. "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other". In: *The Annals of Mathematical Statistics* 18.1 (1947), pp. 50–60. DOI: 10.1214/aoms/1177730491.

[57] Sture Holm. "A Simple Sequentially Rejective Multiple Test Procedure". In: *Scandinavian Journal of Statistics* 6.2 (1979), pp. 65–70. ISSN: 03036898, 14679469. URL: https://www.jstor.org/stable/4615733.

[58] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Benchmark Data Analysis*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/benchmark-data-analysis.

[59] Selenium. *Selenium*. URL: https://www.selenium.dev (visited on 06/06/2026).

[60] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Benchmark Runner*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/benchmark-runner.

[61] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Process-Measuring Tool*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/process-measuring-tool.

[62] Devographics. *State of JavaScript*. 2025. URL: https://2025.stateofjs.com.

[63] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Database Seed*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/database-seed.

[64] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *FrameworkBenchFullStack Replication Package*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/replication-package.

[65] Mahmoud A. Bokhari, Brad Alexander, and Markus Wagner. "Towards rigorous validation of energy optimisation experiments". In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. GECCO '20. Association for Computing Machinery, 2020, pp. 1232–1240. DOI: 10.1145/3377930.3390245.

[66] Nicolas Geoffray et al. "VMKit: a substrate for managed runtime environments". English. In: *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Association for Computing Machinery, 2010, pp. 51–61. DOI: 10.1145/1735997.1736006.

[67] Firefox Source Docs. *Firefox Source Docs - Introduction to Marionette*. URL: https://firefox-source-docs.mozilla.org/testing/marionette/Intro.html (visited on 04/12/2026).

[68] Vue.js. *Composition API: Lifecycle Hooks | Vue.js*. URL: https://vuejs.org/api/composition-api-lifecycle.html (visited on 06/05/2026).

[69] React. *Queueing a Series of State Updates*. URL: https://react.dev/learn/queueing-a-series-of-state-updates (visited on 06/05/2026).

[70] Ruby on Rails. *Choosing Between Import Maps and a JavaScript Bundler*. URL: https://guides.rubyonrails.org/working_with_javascript_in_rails.html#choosing-

between-import-maps-and-a-javascript-bundler (visited on 06/05/2026).

[71] Sustainable Web Design. *Estimating Digital Emissions*. URL: https://sustainablewebdesign.org/estimating-digital-emissions/ (visited on 06/04/2026).

[72] Qwik. *Qwik - Deliver instant apps at scale*. URL: https://qwik.dev (visited on 06/06/2026).

[73] Khaoula Benmoussa et al. "A new model for the selection of web development frameworks: application to PHP frameworks". In: *International Journal of Electrical and Computer Engineering* 9.1 (2019), p. 695. DOI: 10.11591/ijece.v9i1.pp695-703.

[74] Wappalyzer. *Wappalyzer Chrome Extension*. URL: https://www.wappalyzer.com/apps/ (visited on 05/13/2026).

[75] Jürgen Börstler et al. "Developers talking about code quality". In: *Empirical Software Engineering* 28.6 (2023). DOI: 10.1007/s10664-023-10381-0.

[76] Luis Cruz et al. "Do Energy-Oriented Changes Hinder Maintainability?" In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 29–40. DOI: 10.1109/ICSME.2019.00013.

[77] Yoojin Choi et al. *Jovis: A Visualization Tool for PostgreSQL Query Optimizer*. 2025. DOI: 10.48550/arXiv.2411.14788. arXiv: 2411.14788 [cs.DB].

[78] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Site Verifier*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/site-verifier.

[79] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Benchmark Runner*. URL: https://github.com/FrameworkBench-RepPack/benchmark-runner/ (visited on 04/27/2026).

[80] Apple. *Recount - Apple-Oss-distributions/XNU*. URL: https://github.com/apple-oss-distributions/xnu/blob/f6217f891ac0bb64f3d375211650a4c1ff8ca1ea/doc/observability/recount.md.

[81] Kashif Nizam Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3.2 (2018). DOI: 10.1145/3177754.

[82] Firefox DevTools. *Firefox Devtools - Gecko profile format*. 2023. URL: https://github.com/firefox-devtools/profiler/blob/354d108db40275f806775b8c63b4069a84584678/docs-developer/gecko-profile-format.md (visited on 04/27/2026).

[83] W3C. *WebDriver*. URL: https://www.w3.org/TR/webdriver2/ (visited on 04/27/2026).

[84] Selenium. *Firefox specific functionality | Selenium*. 2025. URL: https://www.selenium.dev/documentation/webdriver/browsers/firefox/#context (visited on 04/27/2026).

[85] Albert Hald Christensen, Anton Juul-Naber, and Bastian Brix Hansen. *Benchmark Runner*. 2026. URL: https://github.com/FrameworkBenchFullStack-RepPack/benchmark-runner-data-processor.

[86] Firefox DevTools. *Firefox Devtools - Firefox Profiles Types*. 2026. URL: https://github.com/firefox-devtools/profiler/blob/a887d1c0b3ecf87253a8b5cdaa491fa5aee860be/src/types/gecko-profile.ts (visited on 05/28/2026).

APPENDIX

APPENDIX A: TEST SITES

1. Test Site Design

To determine a relevant test site design, an analysis of existing websites is carried out. The goal of this analysis is to identify websites that use full-stack frameworks, categorize them based on their content and purpose, and finally determine common elements across those categories. Elements can be website features such as lists, navigation bars, or boxes with form filters.

1) Full-Stack Framework-Based Websites: To discover high-impact websites that use full-stack frameworks, the most visited websites are examined. The list of most popular websites is sourced from [7], which used it to categorize websites and determine common website elements. All of the websites are re-examined to remove those that do not use full-stack frameworks. The frameworks of the remaining websites are noted. The utilized frameworks are identified with the help of the browser's developer tools and a browser extension called Wappalyzer [74]. Wappalyzer analyzes website content and network requests to determine which technologies are used to build it. In addition to their framework, they are once again categorized based on their content. While a majority of categories are reused from the original list, a few redundant categories have been removed. The list in Table XVII shows all the sites that use a full-stack framework, as well as their categorization.

Website Name	Classification(s)	Full-stack Framework
chatgpt.com	Messaging	Next.js
reddit.com	Forum	Django with React
amazon.com	Storefront	Spring
weather.com	Analysis tool, News	Flask
msn.com	News	ASP.NET Core
linkedin.com	Social media	Play
microsoft.com	Organization, Storefront	ASP.NET Core
netflix.com	Video	Spring Boot
live.com	Email	ASP.NET Core
pinterest.com	Social media	Django and Flask
twitch.tv	Video	Ruby on Rails
naver.com	News	Spring Boot
openai.com	Messaging	Next.js
canva.com	Advanced editor, Organization	Spring Boot
sharepoint.com	Advanced editor, Organization	ASP.NET Core
office.com	Advanced editor, Organization	ASP.NET Core
aliexpress.com	Storefront	Spring Boot
paypal.com	Organization, Documentation	Spring, swapped to Express.js
github.com	Organization	Ruby on Rails
discord.com	Messaging, Organization	Flask
spotify.com	Organization	Spring
vk.com	Social media	Edited Symfony with KPHP
globo.com	News	Ruby on Rails
ebay.com	Storefront	Spring Boot with Marko front-end
walmart.com	Storefront	Hapi.js
booking.com	Storefront	Spring Boot
chaturbate.com	Video	Django
chess.com	News, Forum, Video, Documentation	Symfony
amazon.co.jp	Storefront	Spring
amazonaws.com	Organization	Spring
quora.com	Forum	Pylons
espn.com	News	Next.js
amazon.de	Storefront	Spring
cnet.com	News	ASP.NET Core
spiegel.de	News	Spring Boot
bbc.com	News	Spring
bbc.co.uk	News	Spring
indeed.com	Job index	Spring Boot
cnn.com	News	Stellar
rakuten.co.jp	Storefront	ASP.NET Core
onlyfans.com	Social media	Ruby on Rails
usps.com	Organization, Storefront	Spring Boot
adobe.com	Organization	Apache Sling

Continued on next page

Website Name	**Classification(s)**	**Full-stack Framework**
uol.com.br	News	Spring Boot
etsy.com	Storefront	Spring with Scala
shein.com	Storefront	Symfony
accuweather.com	Analysis tool, News	Spring Boot
amazon.co.uk	Storefront	Spring
primevideo.com	Video	Spring Boot
ozon.ru	Storefront	ASP.NET Core
dzen.ru	News, Video	ASP.NET Core
marca.com	News	Proprietary CMS
instructure.com	Organization	Ruby on Rails
tradingview.com	Analysis tool, Documentation, Organization	Django
shopify.com	Organization	Ruby on Rails
rutube.ru	Video	Mobx.js
moloco.com	Organization	Spring Boot
infobae.com	News	Arc XP
ikea.com	Organization, Storefront	Spring Boot
disneyplus.com	Video	Spring Boot
amazon.in	Storefront	Spring
theguardian.com	News	Play
amazon.it	Storefront	Spring
mercadolivre.com.br	Storefront	Spring Boot
scribd.com	Archive	Ruby on Rails

TABLE XVII: List of the most visited full-stack framework-based websites, their used framework, and their categorization.

To get an overview of the utilized full-stack frameworks, their entries are counted and listed in Table XVIII, ranked by popularity.

Framework	Occurrences
Spring Boot	16
Spring	12
ASP.NET Core	9
Ruby On Rails	7
Django	4
Next.js	3
Symfony	3
Flask	2
Play	2
Pylons	1
Hapi.js	1
Stellar	1
Apache Sling	1
Mobx.js	1
Arc XP	1

TABLE XVIII: Full-stack framework and their number of occurrence in Table XVII.

The identified categories are listed in Table XIX along with website elements that are commonly found on those websites. The categories are derived from the respective websites' content and are used to guide the design of the test site, which will use multiple subpages to represent all of the identified categories.

2) Designing the Test Site: To ensure that the test site is representative of the most visited full-stack framework-based websites, it must represent all of the identified categories. The most common elements were navigation bars and footers, which will be added to all of the test site subpages.

To represent the static elements present on forums, news sites, and archive websites, the test site will include two subpages, *static-1* and *static-2*, both of which contain only static text. A wireframe representing the structure of these subpages can be seen in Figure 2.

Category	Elements
Messaging	Live messages and a list of chats or people
Forum	List of questions and answers
Video	List of videos and filters for finding them
Storefront	List of products and filters for finding them
Social Media	List of posts and live messaging
Organization	Static pages with text
News	List of static pages with text
Advanced Editor	A canvas within the website that allows for advanced actions
Archive	Static pages of text

TABLE XIX: The categories and elements discovered as part of the analysis of the most-visited websites that use full-stack frameworks.

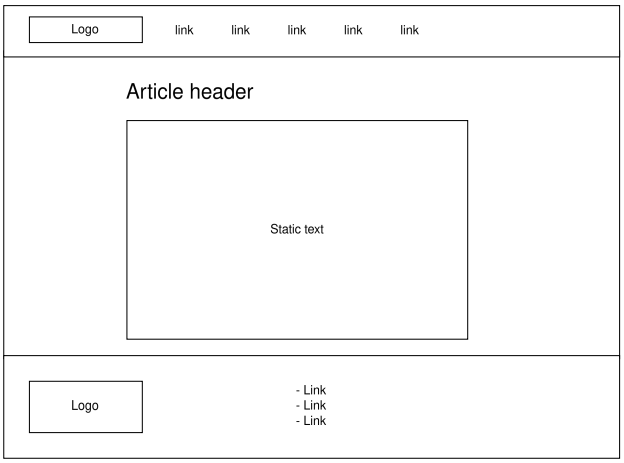


Fig. 2: Wireframe for the design of the static subpage on the test website. The figure is from [7].

The main content of the static pages consists of a title followed by multiple paragraphs of text. The goal is to represent the simplest of websites that do not require much work from the full-stack framework.

To represent the messaging websites, the *live* subpage will contain live data being streamed from the server to the client. The wireframe is shown in Figure 3.

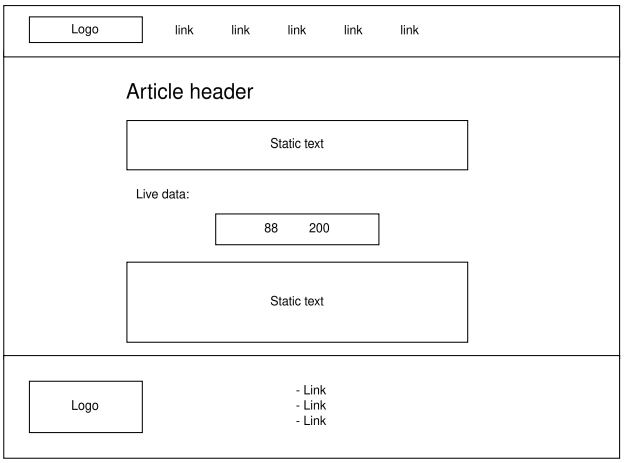


Fig. 3: Wireframe for the design of the live subpage on the test website. The figure is from [7].

The *list* subpage is designed to cover everything from web-shops to social media sites. The pattern discovered from analyzing the websites is that many use cases can be simplified to having lists of content. Video sites have videos, news sites have articles, and stores have products. The *list* page will emulate this by rendering a large list of items with a filter

to alter the list. A wireframe of the list subpage can be seen in Figure 4.

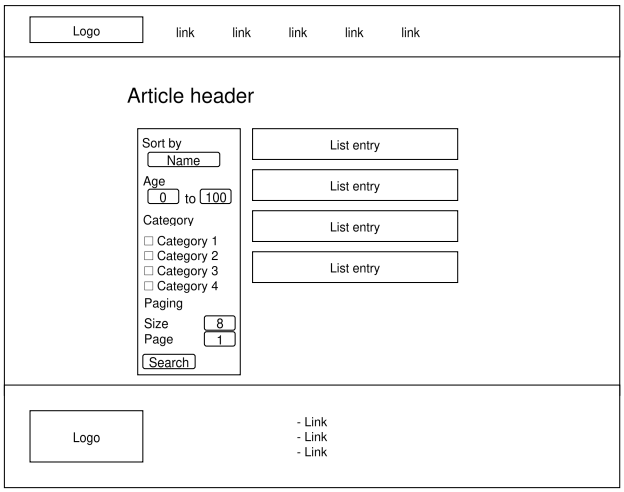


Fig. 4: Wireframe for the design of the list subpage on the test website. It was originally designed in [7] and has been adapted for use in this project.

The *home* page, shown in Figure 5, represents rich social media websites, which contain both lists of posts and messaging features. It does this by combining the content of the other pages such that it includes both live data and a filterable list.

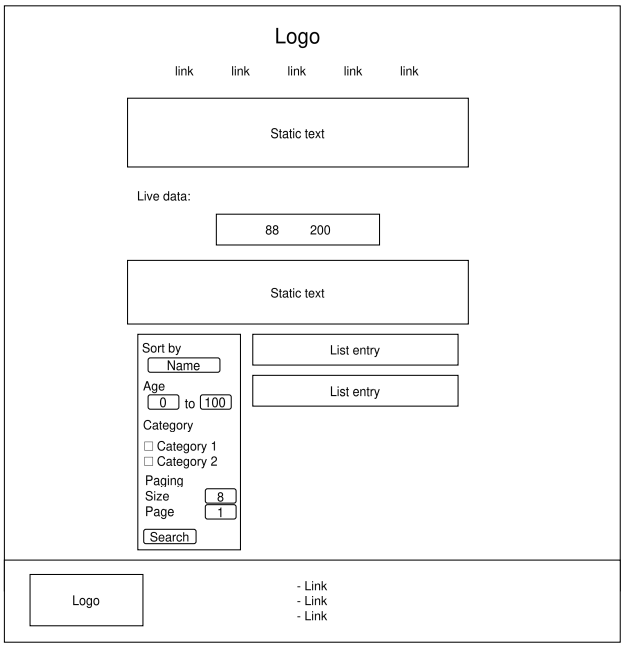


Fig. 5: Wireframe for the design of the home subpage on the test website. It was originally designed in [7] and has been adapted for use in this project.

The goal of this page is to challenge the framework, with both sending live data and having to process list data.

2. Test Site Implementation

While developing software, several implementation strategies can be employed. One of these is to highly optimize the code to ensure that it is as efficient as possible. In [75], 34 professional developers, software teachers, and students were interviewed about their thoughts on code quality. The study found that a majority prioritize traits such as comprehensibility and maintainability over dynamic runtime behavior when evaluating code quality.

As dynamic runtime behavior-focused changes, such as performance improvements, have been found to decrease maintainability [76], the test site implementations will follow the individual framework's recommended practices and will not be optimized for either performance or energy efficiency.

In [7], development guidelines were used to facilitate similar development practices. While a majority of these guidelines are also applicable for full-stack frameworks, their linting-related guideline cannot be reused, as not all frameworks provide official linters. The remaining guidelines have been adapted, and additional full-stack framework-specific guidelines relating to their ORMs and the utilized DBMS have been introduced. The complete list of guidelines is listed below:

Guideline 1. Follow recommended practices and commonly used patterns from the framework's official documentation when they are applicable.

- **(a)** Only apply energy-related optimizations when explicitly recommended by the framework's official documentation.

Guideline 2. Prioritize comprehensible and maintainable code over code optimized for performance and energy efficiency.

Guideline 3. Use framework-provided components and methods over custom implementations.

Guideline 4. Use the ORMs and DBMSs to handle the operations for which they are designed, rather than performing them manually.

Guideline 5. All test sites must be structurally and visibly identical to the reference site, except for cases where framework-provided components create structurally different HTML.

Guideline 6. Use the most recent stable software versions.

(a) Only use stable browser APIs that are supported by all major browsers.

Guideline 7. Before finishing an implementation, at least one other group member must review the whole implementation, ensuring that potential implementation differences or bugs are corrected.

In addition to the previously discussed practices, the guidelines also concern developing the test sites in accordance with official documentation and making sure that they are bug-free. Guideline 4 further specifies that data-related operations such as querying, filtering, sorting, and joining must be performed using the ORMs and DBMS. This is to ensure that all implementations have an equal chance to benefit from the highly optimized query planner in the DBMS [77].

1) Caching Policy: Frameworks can set caching policies in the browser, which can influence how much data is fetched when resources are reused across page navigations. To ensure a fair comparison, all frameworks are configured to follow the same guidelines. All resources are cached for at least one hour, which is much longer than the longest test duration of 22 seconds discovered during preliminary tests. The connection to fetch live data is explicitly not cached. Some frameworks provide immutable network resources, meaning that if the resource ever changes, the name also changes. These frameworks are allowed to request immutable caching, which tells the browser to cache the resource forever and never revalidate it.

2) Site Verifier: To help ensure that Guideline 5 is followed and that the correct caching policies are utilized for each web request, a site verifier is developed. Similar to the benchmark runner, the site verifier uses Selenium for automating browser interactions. It can navigate to the test site's pages, where it is able to validate whether the form works as expected and whether the list content is filtered and sorted correctly.

The verifier checks the following:

1) The list is structured correctly and behaves as expected, including:
   a) Default contents
   b) Age filtering
   c) Empty and invalid inputs
   d) Pagination
   e) Parsing of URL-encoded search parameters
   f) Correct HTML structure
2) The live subpage displays the correct values in the expected order.
3) The server sets the expected Content Security Policy (CSP), Cross-Origin Resource Sharing (CORS), X-Content-Type-Options, and Cache-Control response headers.
4) The page uses the correct meta-tags and expected DOM structure.

One of the benefits of using SSR, which is facilitated by all of the tested full-stack frameworks, is that it enables the website to run on devices with JS disabled. The site-verifier ensures that all of the test site implementations support this by performing each check twice: once with JS enabled and once with JS disabled. Its implementation can be found on GitHub [78].

APPENDIX B: MEASURING SERVER PROCESS

In [7], only the client-side energy consumption is measured. To automate the benchmarking process and avoid requiring human interaction, a tool called benchmark runner was developed and used [79]. To ensure that each framework was tested under equal conditions, its test sites were exported as pure HTML, CSS, and JS and then served locally using an identical server setup. Because full-stack frameworks perform back-end processing while serving the website, this test method cannot be applied to them. In addition to client-side energy consumption, testing full-stack frameworks also requires measuring both back-end energy consumption and bandwidth usage. To facilitate this, a process energy measuring tool is developed, and the benchmark runner is expanded to support this tool.

1. Process-Measuring Tool

The benchmarking tool used in [7] starts the server for each test site, sequentially performs all tests, then shuts down the server before proceeding to the next test site. To start and stop measurements of the server process, the tool must communicate with the benchmark runner. This is required to isolate energy consumption to the test runs and exclude the energy consumption associated with the server's initial start-up cost. Additionally, to avoid blocking execution of critical threads such as the measurement or server process thread, the internal and external communication approach must be carefully considered. An overview of the tool's architecture can be seen in Figure 6.

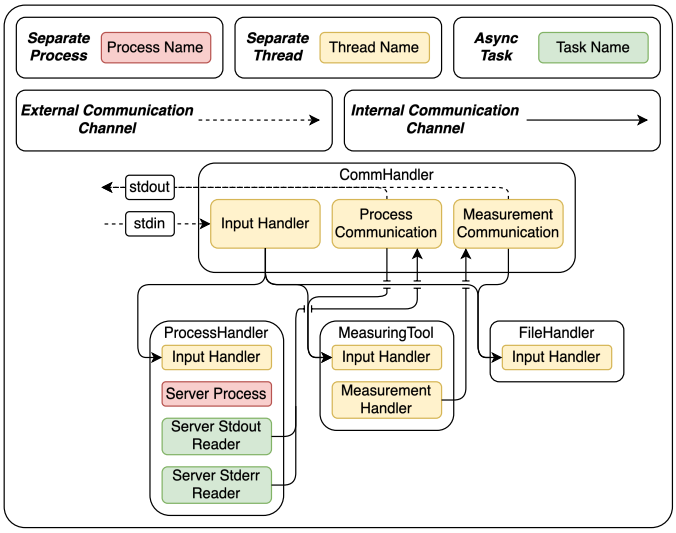


Fig. 6: ProcessMeasuringTool communication flow.

1) Measuring Energy Consumption: The energy consumption measurement method used in [7] uses the Gecko Profiler available in Firefox. When running on Apple Silicon-based Macs, the Gecko Profiler sources its energy measurements from Recount, the resource accounting subsystem of the XNU kernel in macOS [80]. When utilizing Recount, the Gecko Profiler is able to obtain per-thread energy measurements and thereby isolate the energy consumed by, e.g., a single browser tab. This is facilitated by Recount's ability to attribute energy consumption to individual processes and threads at thread context switches.

A widely used alternative to Recount is Intel RAPL, which, similarly to Recount, works by maintaining hardware energy counters in the processor. Intel RAPL enables retrieving the energy consumption at the granularity of the attached DRAM, the entire processor package, its cores, and the iGPU [81]. Due to RAPL being restricted to these domains, it is not possible to extract process-isolated energy consumption measurements. To overcome this limitation, recent literature has presented multiple energy attribution methods such as Scaphandre [21], EnergAt [22], and METRION [23]. Scaphandre and EnergAt operate by distributing the measured energy consumption among processes and threads based on their CPU-time. METRION, on the other hand use this technique only for idle-power distribution while depending on Hardware Performance Counters (HWPCs) for the remaining energy consumption.

To avoid potential estimation inaccuracies as well as to be able to compare client-side and server-side energy consumption, the measurements for the back-end will also be sourced from Recount.

Recount tracks the energy consumption of every process at the granularity of thread context switches. At each switch, Recount takes a snapshot of the specific energy consumption counter, after which it calculates the difference from the previous snapshot. This difference is then scaled to nanojoules, and finally attributed to the specific process [80]. To leverage Recount for the back-end process, the tool interacts with kernel-exposed OS interfaces. The Gecko Profiler directly uses the `<mach/mach.h>` interface to call the provided `task_info` function. This function takes as an argument the Mach task port[1] of the process for which information should be returned. Getting the Mach task port of another process based on its PID is possible but requires strict security privileges, so this approach cannot be utilized for the back-end process. The same energy information can be obtained by utilizing the `proc_pid_rusage` function from the process-information interface `<libproc.h>`. For a given process, the function returns the cumulative energy consumption in nanojoules, among other things.

Preliminary tests reveal that some frameworks, such as ASP.NET Core and Ruby on Rails, unexpectedly consume very low amounts of energy. After further investigation, it is discovered that they use separate subprocesses for part of their processing. This is confirmed by checking the activity monitor, which shows that the measured PID is the parent process for one or more child processes. As the measured energy consumption of a process only includes the consumption of its threads and not separate child processes, the process-measuring tool must account for this.

To calculate the total energy consumption of all sub-processes, the processing tool first has to obtain all of their PIDs. To do this, the `proc_listpids` function from `<libproc.h>` can be utilized with the `PROC_PPID_ONLY` type, which returns a list of all PIDs whose parent matches the provided PID. To obtain potential sub-sub-processes, this is done recursively before every measurement. By doing it before every measurement, the tool can measure potential temporary processes that might exist only for a short time. After obtaining the list of PIDs, it extracts energy measurements for each process and then returns the total energy consumption.

2) Spawning the Server Process: The `ProcessHandler` seen in Figure 6 is responsible for starting, stopping, and

[1] A unique identifier for that task.

communicating with the server process. To facilitate spawning and managing another process, three Boost C++ libraries are utilized:

- `filesystem` for resolving the path of the executable.
- `process` for spawning and managing the external process.
- `asio` for asynchronously managing I/O pipes to the spawned process.

Boost libraries are used for their portability. For example, `boost::filesystem` provides platform-independent path handling, whereas `boost::process` provides a platform-independent abstraction layer for spawning and managing external processes.

As seen in Figure 6, the `ProcessHandler` combines multiple concurrency mechanisms. The server is launched as a separate process while asynchronous read operations are registered for its stdout and stderr channels. These operations are invoked when the `boost::asio` event loop detects a complete line on the respective output channel. Additionally, a dedicated thread is used for handling internal messages from other parts of the tool. To handle cases where the server process quits unexpectedly, a listener is registered to perform a cleanup procedure and stop the remaining threads and tasks.

3) Managing Thread Communication: To facilitate non-blocking, thread-safe communication throughout the tool, multiple `InternalMessageChannel` instances are used. Each instance is in Figure 6 depicted with a solid arrow. As seen in Listing 1, `InternalMessageChannel` instances comprise a queue, a mutex lock, and a condition variable, as well as three methods which ensure thread-safe access to the queue.

```
template <class T>
class InternalMessageChannel {
   std::mutex mutex;
   std::condition_variable cond_var;
   std::queue<T> queue;

public:
   void add_message(T value)
   T get_message()
   std::optional<T> try_get_message()
};
```

Listing 1: Simplified InternalMessageChannel structure based on code available in /communication/internal-message-channel.hpp

It is the responsibility of the CommHandler to handle communication across internal threads as well as incoming messages from stdin. As seen in Figure 6, it takes external messages, parses and converts them to internal messages, and finally distributes them to their corresponding internal message channels. In addition to its input handler, separate threads are maintained for handling the output channels of the `ProcessHandler` and the `MeasuringTool`. These threads are responsible for distributing messages to the respective internal message channels as well as to stdout.

External communication with the benchmark runner is carried out using JavaScript Object Notation (JSON) formatted messages. These messages contain a `type` entry depicting the message type as well as an optional `payload`. The following message types exist: `measurement_ready`, `measurement_start`, `measurement_stop`, `terminate`, and `set_output_path`. The payload entry is currently only utilized with the `set_output_path` message type, where it contains the new measurement output path. These messages are shared using the process-measuring tool's standard input and output channels.

2. Benchmark Runner

The benchmark runner facilitates performing the benchmarks by automatically orchestrating the database, server, and browser. This subsection details important parts of the benchmark runner, such as how it interacts with the Firefox browser, how it manages the interactions with the database and server, and finally demonstrates how it might communicate with the process-measuring tool in a benchmarking scenario.

1) Programmatically Controlling the Gecko Profiler: Programmatically being able to interact with the browser is important to avoid the threat to validity mentioned in [6] of manually having to start automation scripts and operate the Gecko Profiler. Programmatic interaction with the Gecko Profiler requires running code directly in Firefox's browser chrome, i.e., the browser's own UI [82]. Switching to the browser chrome's context is not officially supported in the W3C WebDriver specification, which is the official specification for the WebDriver remote control interface [83]. By using the geckodriver, a proxy that translates WebDriver requests to Gecko's internal Marionette protocol, the browser automation tool Selenium is able to switch between contexts [84]. It is therefore used to interact and control the browser instance.

The benchmark runner switches to the browser chrome's

context by using the `setContext` function provided by Selenium. It then executes either `StartProfiler`, `Pause`, `dumpProfileToFileAsync`, or `StopProfiler`, and finally switches back to the original context.

2) Running benchmarks: The benchmark runner automatically identifies the benchmarks by the file extension *.bm.ts*, and checks whether they export a function that matches the expected signature. Each of these files takes as input an object of the type shown in Listing 4.

```
type BenchmarkInput = {
  framework: string;
  iteration: number;
  warmupRound: number;
  resultsPath: string;
  link: string;
  profilerOptions: ProfilerOptions;
  driverOptions?: BuilderOptions;
  setServerResultPath: (path: string)
      ↪ => void;
  startServerMeasurement: () => void;
  stopServerMeasurement: () => void;
};
```

Listing 2: BenhcmarkInput type available in /src/benchmarks/benchmark-types.ts

The benchmark runner's input contains information such as name, iteration, warm-up round, and path used for storing the result files. It also contains profiler options for the Gecko Profiler and driver options used to configure the geckodriver and browser instance. Lastly, it contains callback functions used for process-measuring tool communication. All of the benchmark files use a convenience function called `profilerWrapper`. It takes a majority of the `BenchmarkInput` properties, manages browser and Gecko Profiler configuration, and performs the benchmark. This removes the responsibility of configuring the browser and profiler from the individual benchmarks and ensures a consistent configuration. The `profilerWrapper` takes three additional callback functions, a setup function, a benchmark function, and a tear-down function, which it uses when performing the benchmark.

3) Configuring Database and Servers: Starting the server of each framework requires its own start command, a unique regular expression for detecting server start, and potentially a pre-configuration step. To manage these, a central configuration file is created. This file defines an entry for each server following the structure in Listing 3.

```
export type TestSiteConfig = {
  portIdentifier: string;
  start: CommandType;
  startDetectionRegex:
      ↪ StartDetectionRegex;
  prepare?: CommandType |
      ↪ CommandType[];
};
```

Listing 3: Simplified TestSiteConfig type. Complete type is available in /src/types/test-sites.ts

In this specification, the `start` and optional `prepare` fields specify commands used for respectively starting and preparing the server. The `startDetectionRegex` field specifies a regular expression that is matched against the server's output to check if the server has finished launching. This regular expression is used internally by the process-measuring tool to detect when the server is ready, at which point it initializes the measuring thread and signals to the benchmark runner that it is ready.

The configuration file also contains commands for preparing, starting, and resetting the database, along with regular expressions for detecting when each step has completed. Additionally, a database URL is provided, which is passed to the servers using an environment variable.

Before performing the benchmark, all of the provided preparation commands are executed, ensuring that the servers and database are ready for the benchmark.

4) Managing Server Process: To manage the server process without blocking the benchmark runner's main loop, a separate worker thread is used. This worker thread is responsible for spawning and facilitating communication with the process-measuring tool. To launch the server process, the `spawn` function is used. The code for launching the process-measuring tool can be seen in Listing 4.

```
const serverProcess = spawn(
  data.processMeasurementExecutable,
  [
    `--command=${data.serverCommand}`,
    `--start-regex=${
       data.startDetectionRegex
    }`,
    `--interval=${
       data.measurementInterval
```

```
    }`,
    `--process-dir=${data.siteDir}`,
  ],
  {
    shell: false,
    env: {
      ...process.env,
      ...data.env,
    },
  },
);
```

Listing 4: Code for starting the process-measuring tool available in /src/utilities/server-worker/measuring-server-worker.ts

The worker is passed a data object which contains:

- Path to process-measuring tool
- Server launch command
- Start detection regex
- Measurement interval
- Working directory of the server
- Additional environment variables to be used

As seen in Listing 4, a majority of this information is passed to the process-measuring tool when launched. After spawning the tool, listeners are registered for its `stderr` and `stdout` channels. The messages sent to `stdout` are as mentioned in Appendix B.A.3, JSON-formatted. Hence, the listener for this channel has to parse the given messages and then sends it to the main benchmark runner loop.

In addition to handling the server process, the worker handles communication with the main benchmark runner thread so that it can communicate with the process-measuring tool. To facilitate this, a listener is registered for messages from the main thread. Upon receiving a message, the corresponding JSON-formatted message is created and sent to the process-measuring tool's `stdin` channel.

5) Benchmark Runner Loop: A central loop in the benchmark runner is responsible for performing the selected benchmarks on the chosen test sites, in the correct sequence. This process involves ensuring that the database is in a clean state, that the specific test site server is running and ready to be measured, as well as initializing the benchmark. In Algorithm 1, this benchmark loop is depicted.

Algorithm 1: Run Benchmark Suite

Input: Benchmark options $options$
Output: Benchmark results for all selected test sites and iterations

```
initializeDatabase()
testSites ← getSelectedTestSites(options)
for iteration ← 1 to options.iterations do
    foreach testSite ∈ testSites do
        foreach benchmark ∈ benchmarks do
            server ← startTestSiteServer(options, testSite)
            waitUntilServerIsReady(server)
            for round ← 1 to options.warmupRounds + 1 do
                benchmarkInput ← createBenchmarkInput(options, testSite, iteration, round, server)
                benchmarks ← loadSelectedBenchmarks(options)
                executeBenchmark(benchmark, benchmarkInput)
            terminateServer(server)
            resetDatabase()
```

As mentioned in Section V-E1, warm-up rounds are utilized to compensate for the fact that frameworks using JIT-compiled languages might require multiple iterations before optimizing the code responsible for serving the related requests. This is reflected in the benchmark as the inner for-loop, which performs the same test-case for `options.warmupRounds` before performing the measured round.

To allow for a preparation step before performing the individual test cases, they are each made responsible for starting and stopping the Gecko Profiler and server. To manage the process-measuring tool from within the individual test-cases, they are passed callback functions that send the corresponding messages to the worker thread, which then forwards them to the process-measuring tool.

6) Example Execution: An example message sequence can be seen in Figure 7. This diagram visualizes the sequence of messages sent between the benchmark runner and the process-measuring tool.

An example of benchmark measurements obtained using a single test case is shown in Figure 8. The data for the chart

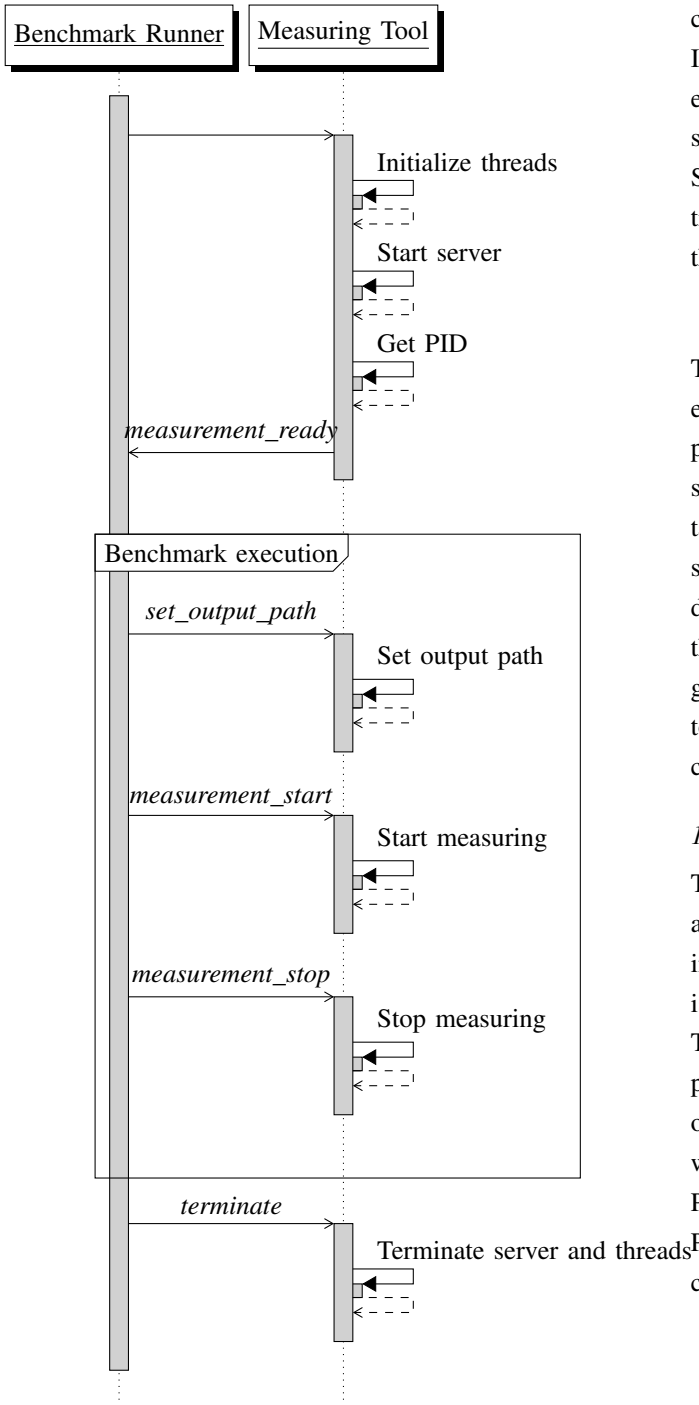


Fig. 7: Benchmark message sequence. The *Benchmark execution* block is repeated for every test case. Solid arrows between the processes depict messages when a *type* is defined over the arrow.

is obtained by performing the `navigate` test case on the `astro-htmx` test site. The chart shows that `astro-htmx` uses two processes named *npm run serve* and *node*. The first process is a utility process which starts the Astro process named *node*. This is reflected in the chart, as the *npm* process contributes relatively little to the overall energy consumption. In contrast, the *node* process consumes significantly more energy, especially when serving the initial request. When serving subsequent requests, its energy consumption decreases. Similarly, the client consumes the most energy during the initial page load compared to subsequent navigations, suggesting that it is able to cache some of the received content.

Appendix C: Processing Measurements

The process-measuring tool and Gecko Profiler outputs their energy measurements in two different formats. While the process-measuring tool exports its measurements as a comma-separated values (CSV) formatted file containing only timestamps and measurements, the Gecko Profiler exports its measurements using a JSON formatted file, which includes a lot of data irrelevant to the study. To extract the measurements from the Gecko Profiler files and to further process both outputs, the gecko-profiler-parser from [7] has been expanded. It is referred to as the benchmark runner data processor and its source code can be found on GitHub [85].

1. Parsing Output Files

The lines of the process-measuring tool's output are formatted as `Time (ms), Energy Consumption (nj)`, allowing it to be parsed line-by-line. The Gecko Profiler output is JSON formatted, as it does not exclusively export energy. This requires us to parse it to extract energy measurements. To parse JSON files, their structure and types must be known. To obtain this information, we have manually inspected the files while cross-examining the types defined in Mozilla's Firefox Profiler web app, which is a tool for visually inspecting Gecko Profiler outputs [86]. The core parts of the resulting schema can be seen in Listing 5.

```
{
  "processes": [{
    "meta": {
      "profilingStartTime": number
    },
    "pages": [{ "url": string, }]
    "counters": [{
      "category": string,
      "samples": {
        "schema": { "<name>": number },
        "data": [[ number ]]
      }
    }]
  }]
}
```

Listing 5: Small part of the Gecko Profiler output's JSON structure

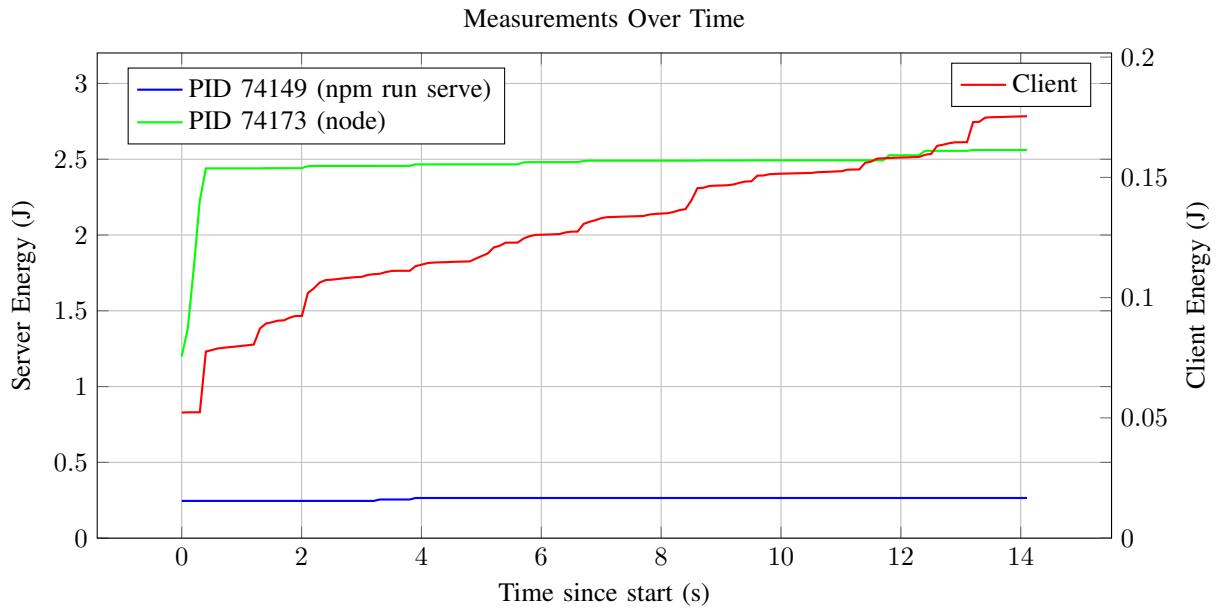


Fig. 8: Accumulated energy consumption of the server and client, obtained while performing the `navigate` test-case on the `astro-htmx` test site.

After parsing the profiler output, the energy measurements are extracted. As the root object contains multiple measured processes, we must identify the one related to the tested tab. To do this, the `pages` property in each `processes` entry is checked for whether one of their URLs is equivalent to the URL used in the benchmark. After identifying the correct process, the `counters` list is checked for an entry where the `category` is `power`. This entry specifies the indexes of `time` and `count` in the `schema` property. By using this information, timestamps and corresponding counts (energy measurements) are extracted from the `data` entry.

2. Processing the Measurements

As the timestamps extracted along with the Gecko Profiler energy measurements reflect the relative time elapsed since profiling start, they are not comparable to the ones from the process-measuring tool. To enable comparison between client and server measurements over time, the Gecko Profiler timestamps must be corrected. Fortunately, as seen in Listing 5, each process contains a property called `profilingStartTime`. When adding this number to the individual measurements, Unix Epoch timestamps are obtained, which are directly comparable to the timestamps from the process-measuring tool.

In addition to correcting the timestamps, the parser ensures that client and server measurements are expressed using the same measurement units. While the Gecko Profiler returns measurements in picowatt-hour (pWh), the process-measuring tool returns them in nanojoules (nJs), which is the unit directly provided by the Operating System (OS). To avoid having to convert their values unnecessarily, they are stored in containers called `PowerAmount`, which keep track of their measurement unit. Upon having to add or read the value, a measurement unit can be provided, and the function automatically converts its value to the given unit. When exporting the measurements, they are retrieved from the containers in Js and exported to the respective files.

APPENDIX D: TEST WARM-UP RESULTS

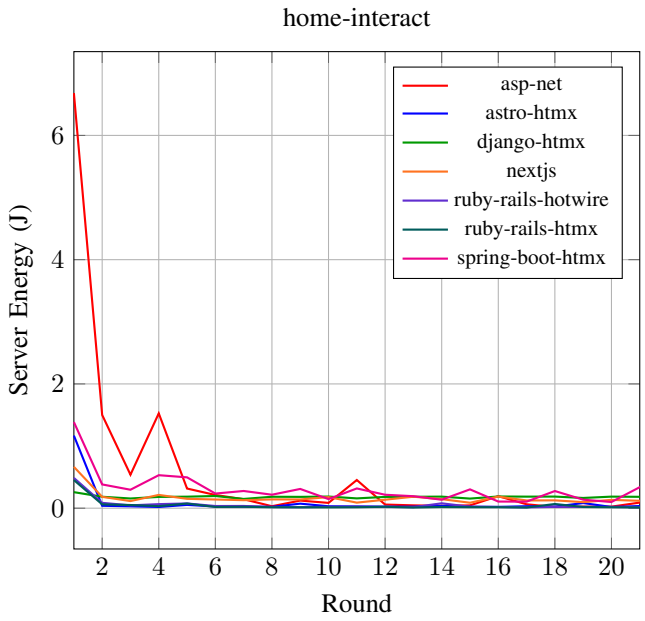


Fig. 9: Chart showing the server-side energy consumption per round from performing the home-interact test case.

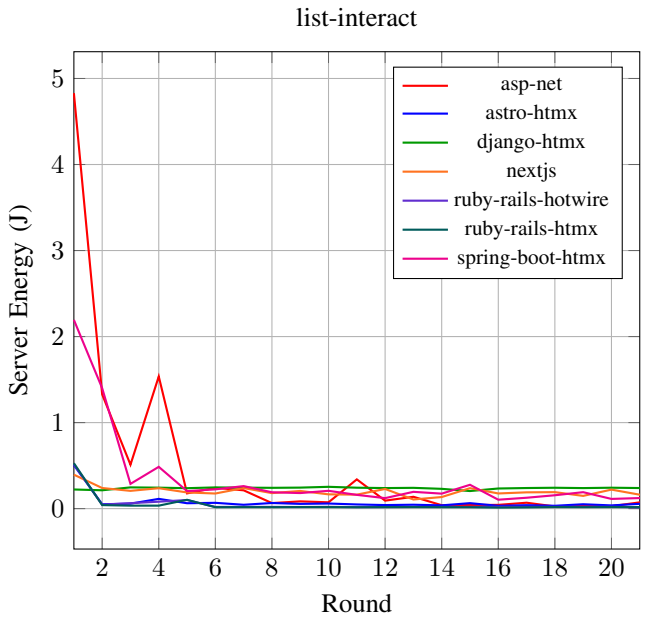


Fig. 10: Chart showing the server-side energy consumption per round from performing the list-interact test case.

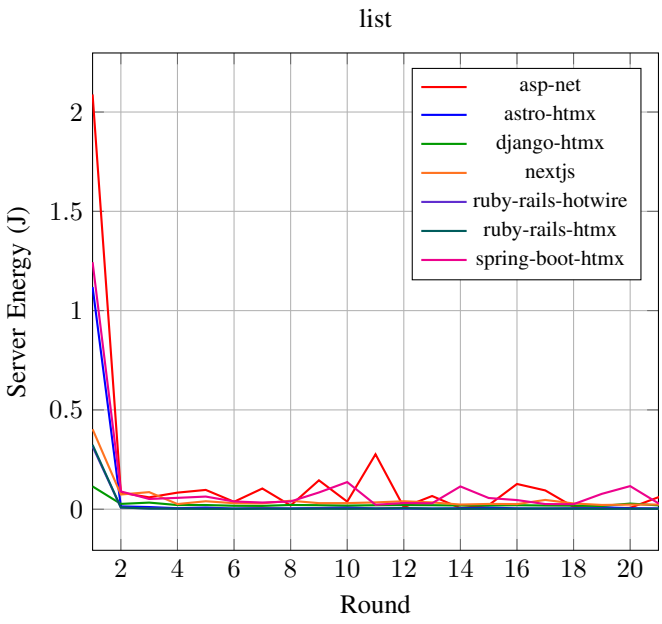


Fig. 11: Chart showing the server-side energy consumption per round from performing the list test case.

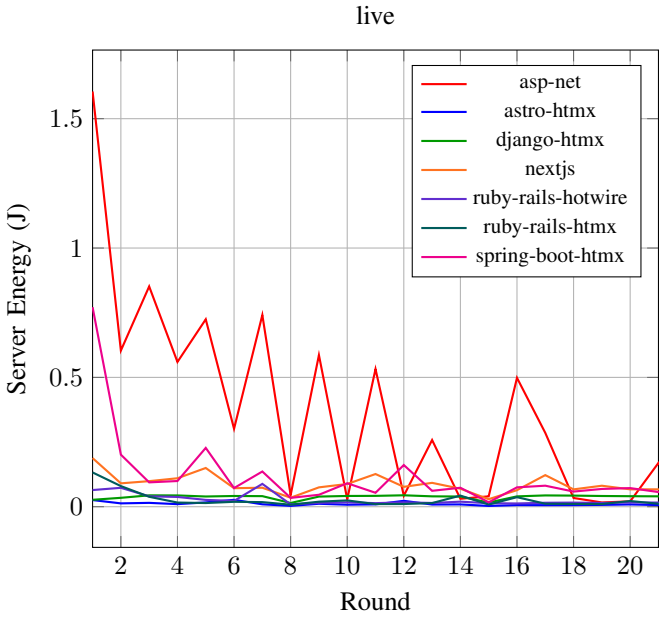


Fig. 12: Chart showing the server-side energy consumption per round from performing the live test case.

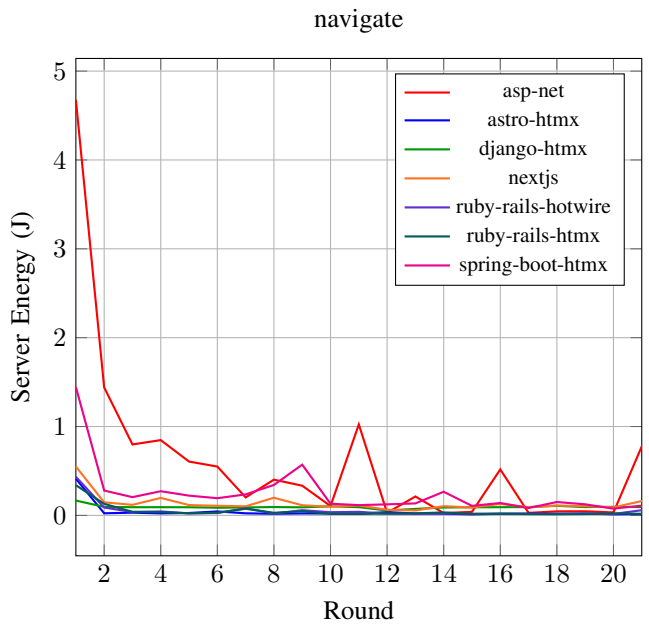


Fig. 13: Chart showing the server-side energy consumption per round from performing the navigate test case.

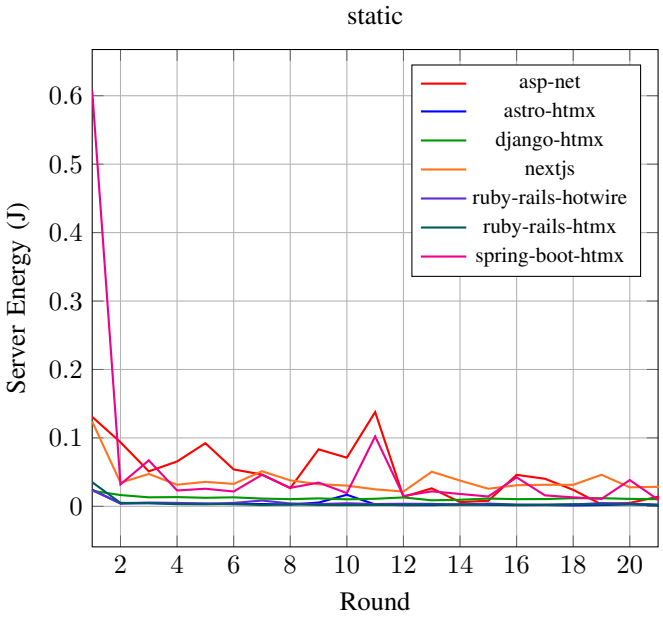


Fig. 14: Chart showing the server-side energy consumption per round from performing the static test case.

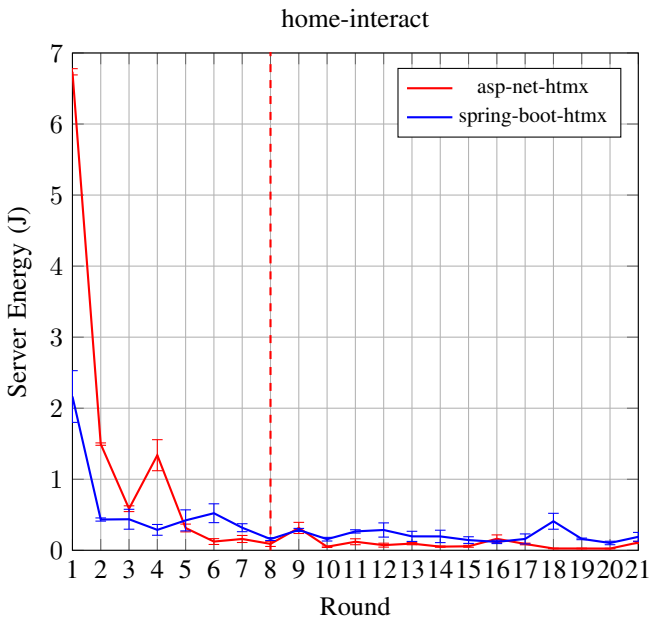


Fig. 15: Chart showing the average energy consumption per round from performing 45 iterations of the home-interact test case on the ASP.NET and Spring Boot test sites. The vertical line depicts the measured round (Warm-up rounds + 1).

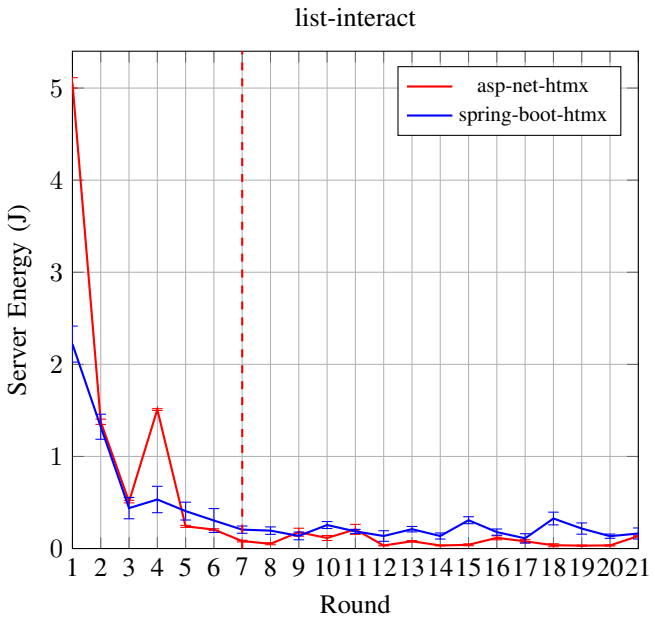


Fig. 16: Chart showing the average energy consumption per round from performing 45 iterations of the list-interact test case on the ASP.NET and Spring Boot test sites. The vertical line depicts the measured round (Warm-up rounds + 1).

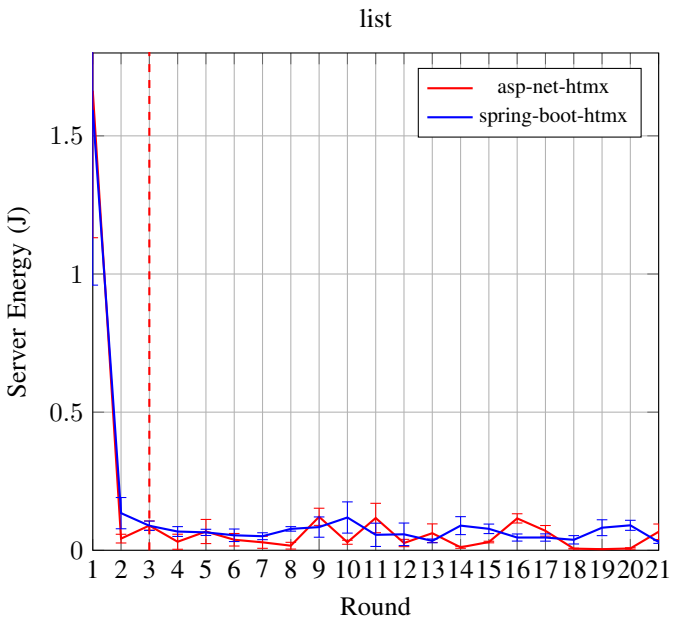


Fig. 17: Chart showing the average energy consumption per round from performing 45 iterations of the list test case on the ASP.NET Core and Spring Boot test sites. The vertical line depicts the measured round (Warm-up rounds + 1).

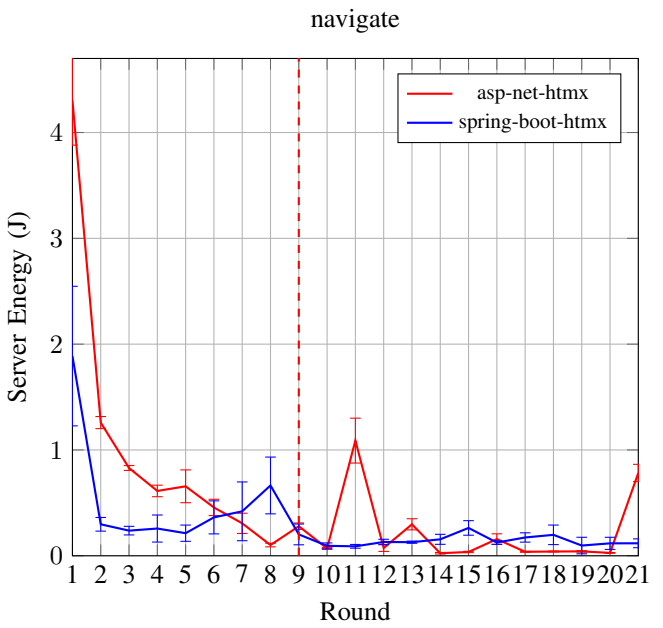


Fig. 19: Chart showing the average energy consumption per round from performing 45 iterations of the navigate test case on the ASP.NET Core and Spring Boot test sites. The vertical line depicts the measured round (Warm-up rounds + 1).

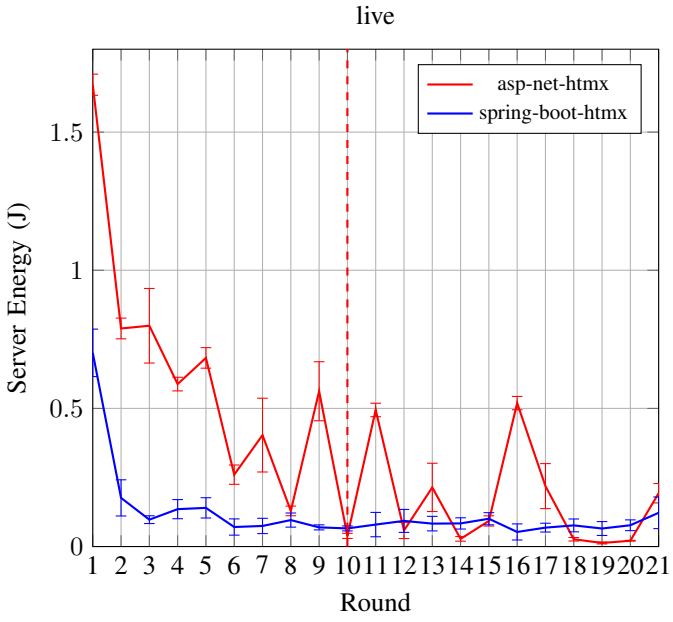


Fig. 18: Chart showing the average energy consumption per round from performing 45 iterations of the live test case on the ASP.NET Core and Spring Boot test sites. The vertical line depicts the measured round (Warm-up rounds + 1).

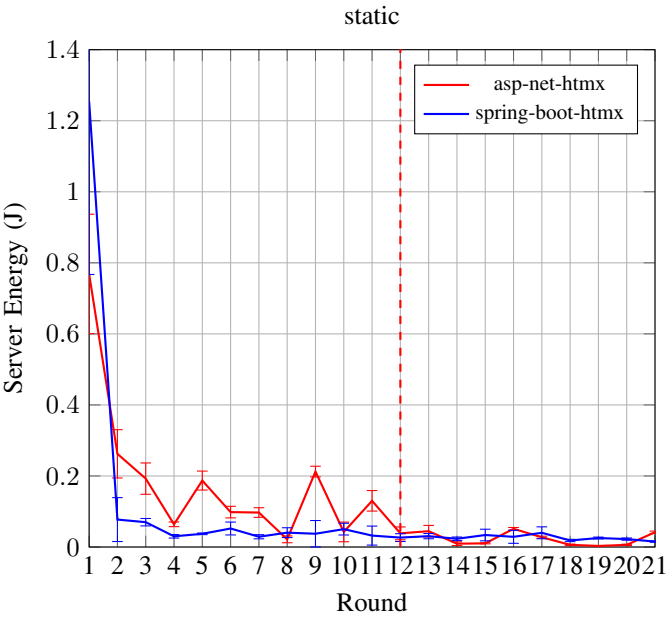


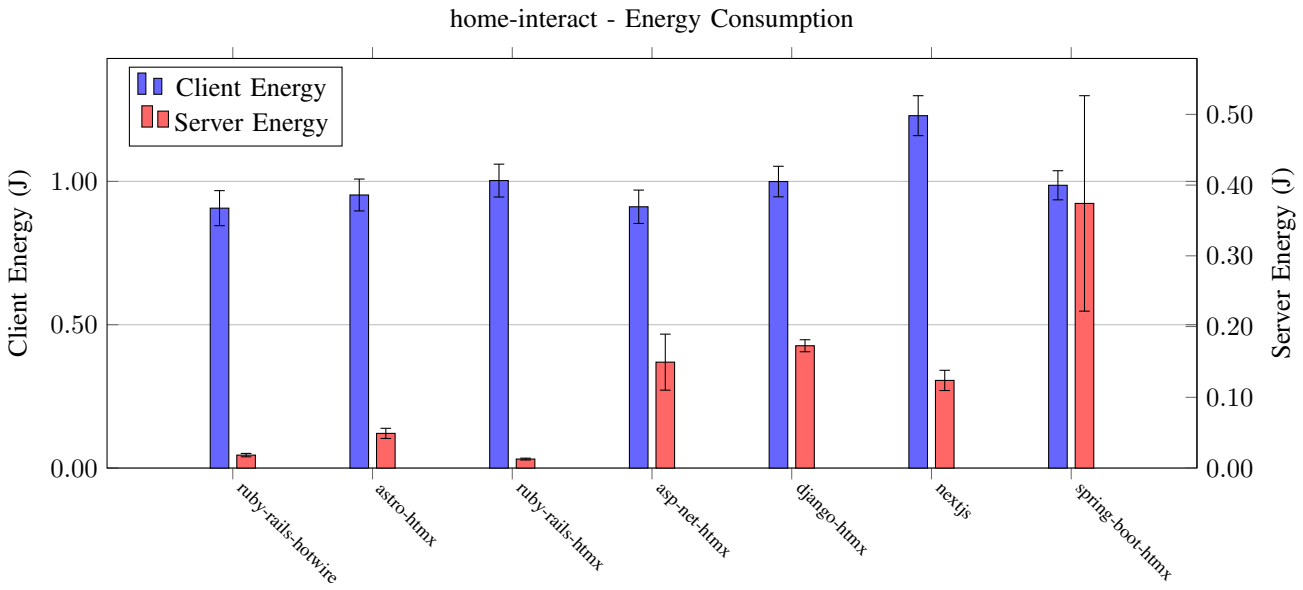
Fig. 20: Chart showing the average energy consumption per round from performing 45 iterations of the static test case on the ASP.NET Core and Spring Boot test sites. The vertical line depicts the measured round (Warm-up rounds + 1).

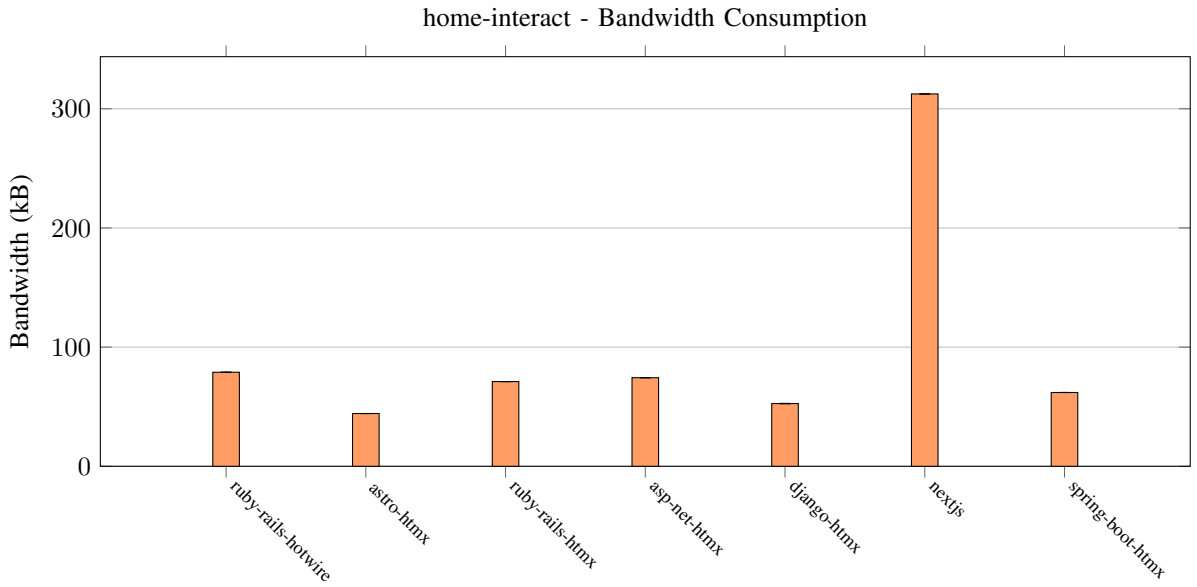
APPENDIX E: TEST RESULTS

This section displays the measurements collected for each test case along with comparison tables depicting the statistical significance of these results.

Each test case is visualized with two graphs and two tables. The first graph shows the client-side and server-side energy consumption separately, while the second graph shows the bandwidth consumption. The tables are followed by two comparison matrices, one for client-side and one for server-side energy consumption. They show which test site results are statistically significantly different when compared to other test sites. The color legends in these matrices mean the following:

- **Red:** the row has statistically significantly higher energy consumption than the column.
- **Blue:** the row has statistically significantly lower energy consumption than the column.
- **Grey:** row and column have no statistically significant difference.





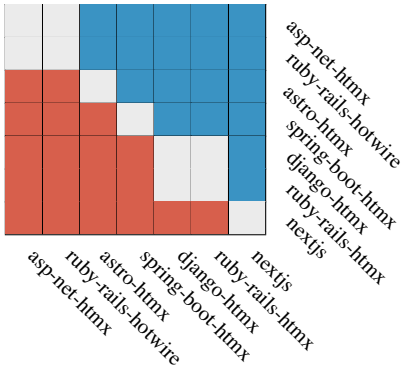


TABLE XX: Statistical comparisons - `home-interact` - Client energy.

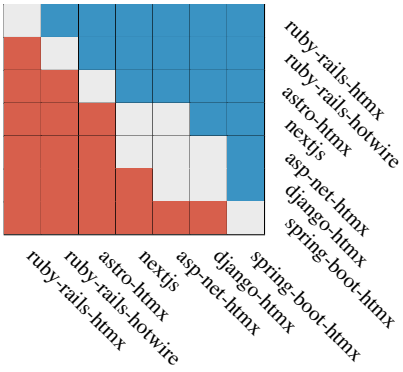
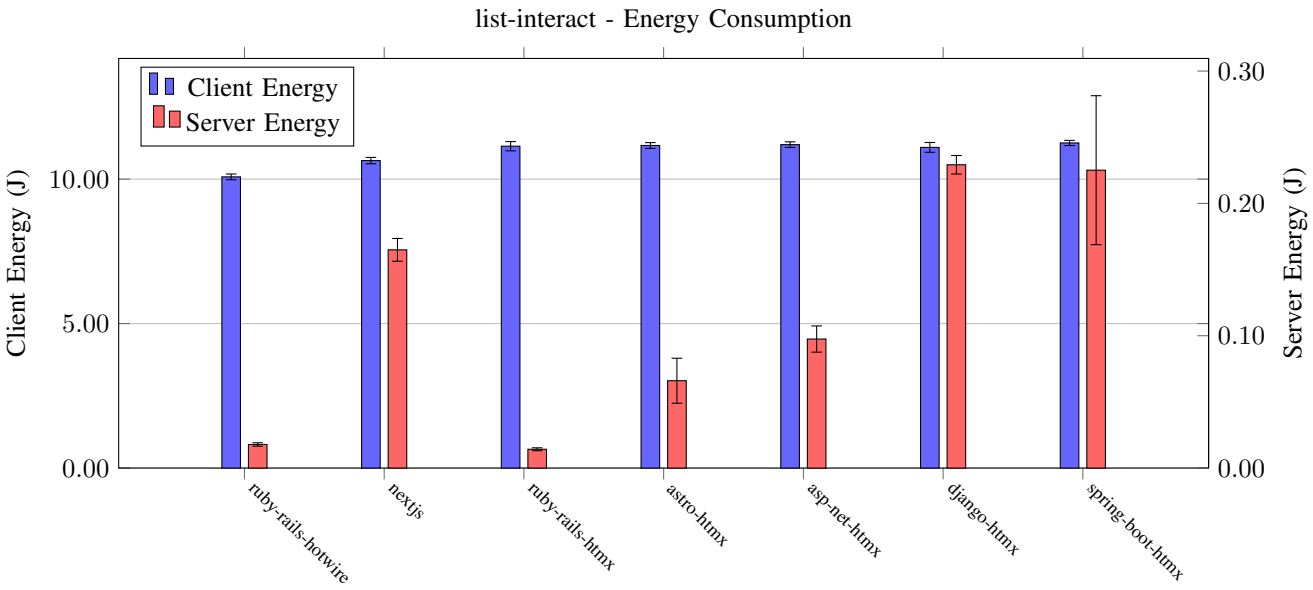
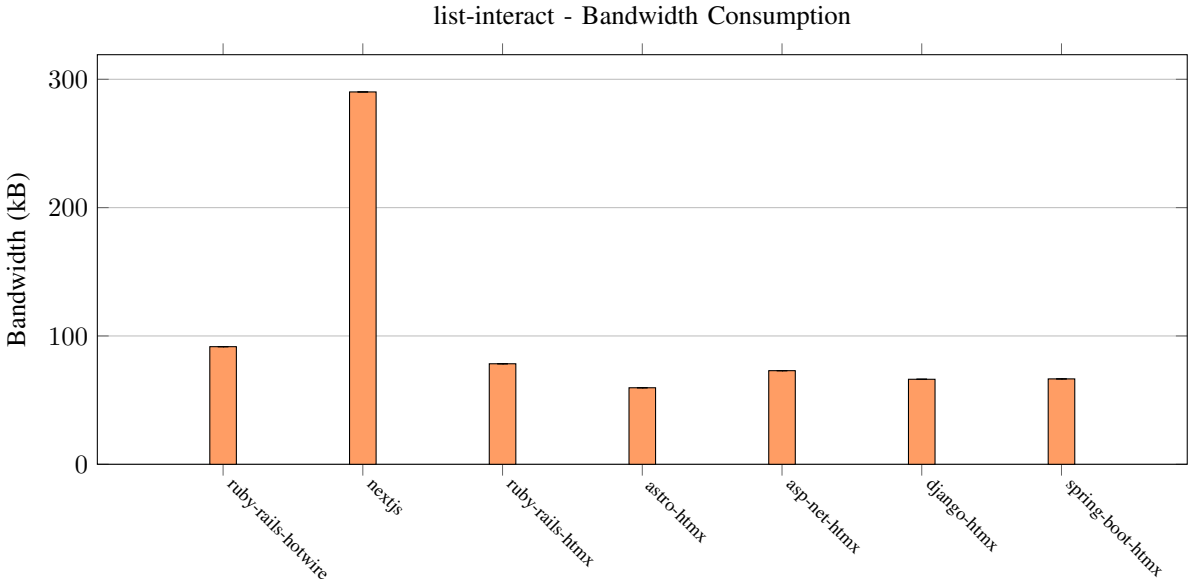


TABLE XXI: Statistical comparisons - `home-interact` - Server energy.





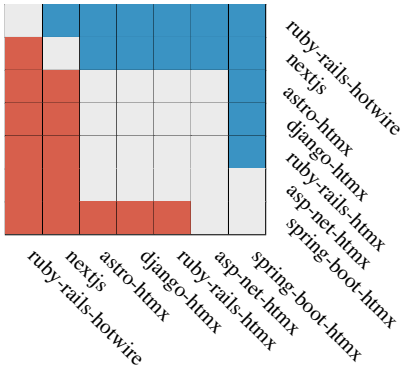


TABLE XXII: Statistical comparisons - `list-interact` - Client energy.

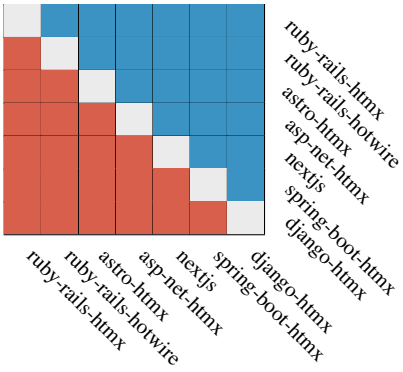
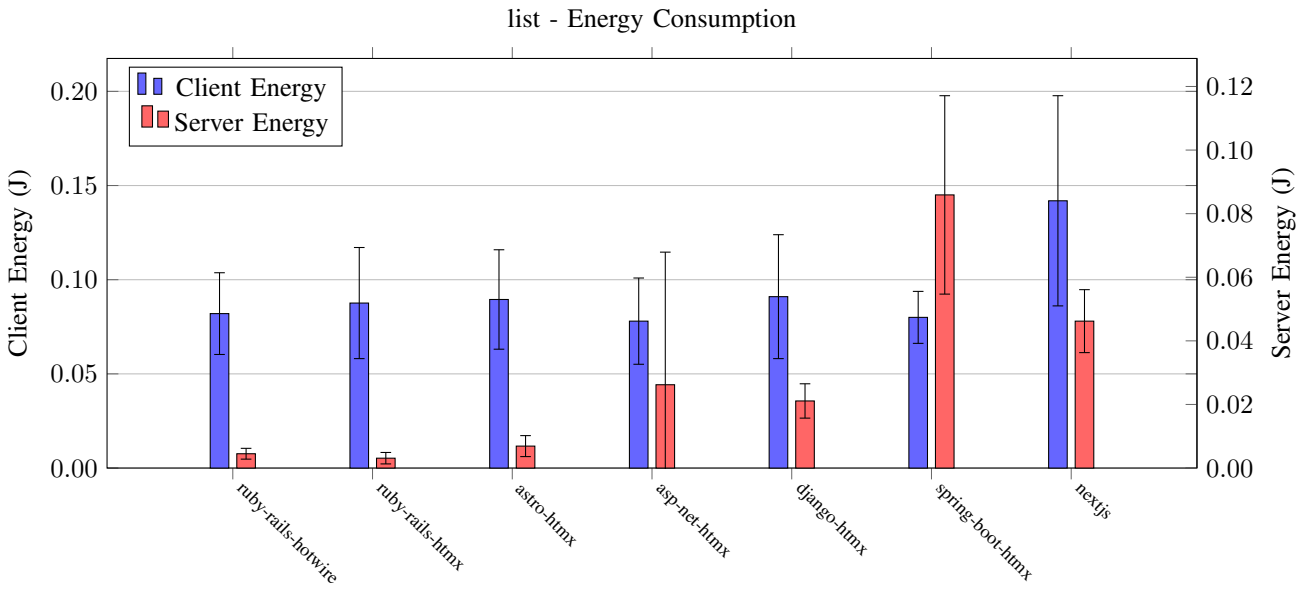
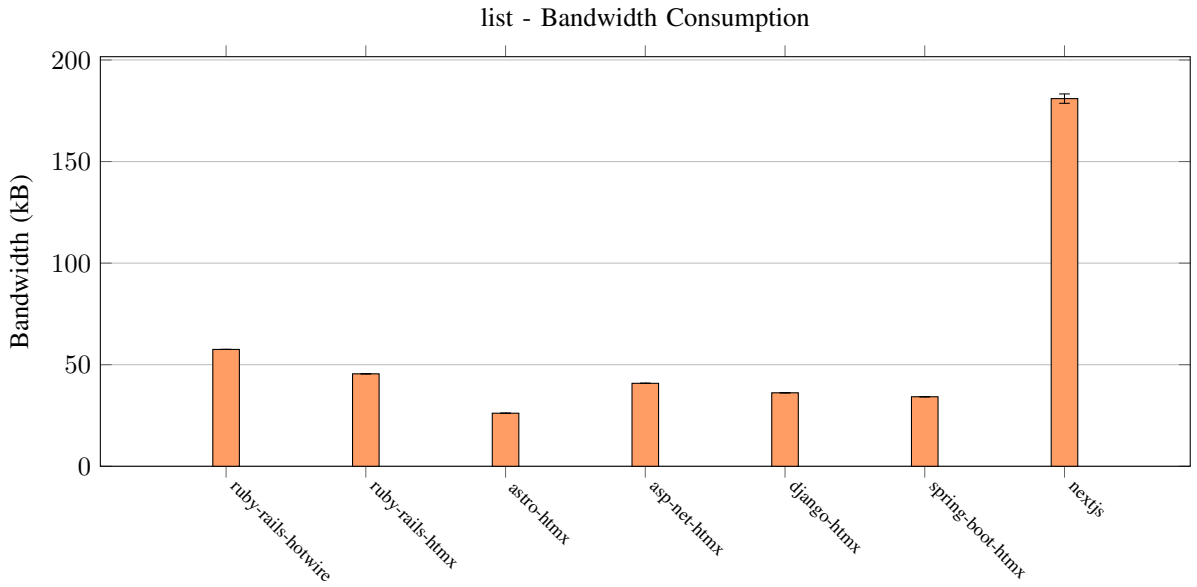


TABLE XXIII: Statistical comparisons - `list-interact` - Server energy.





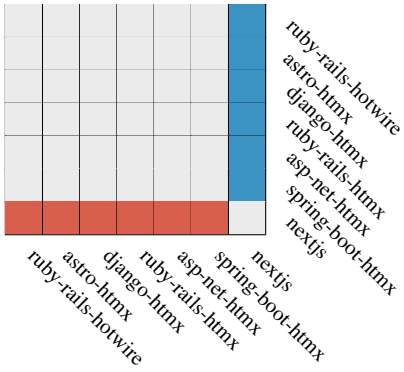


TABLE XXIV: Statistical comparisons - `list` - Client energy.

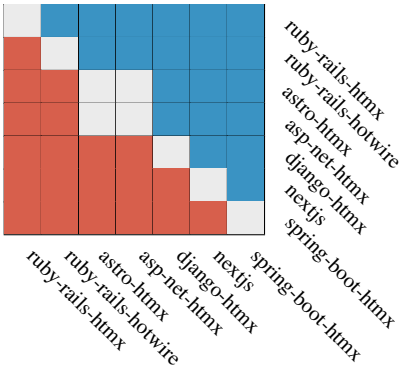
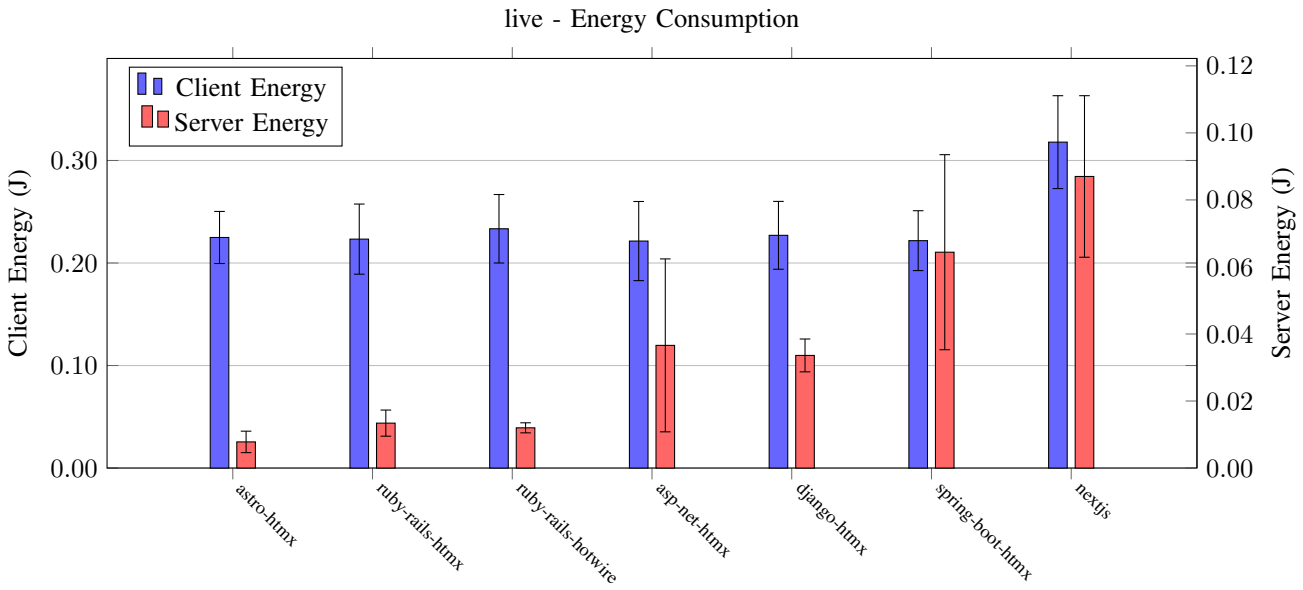
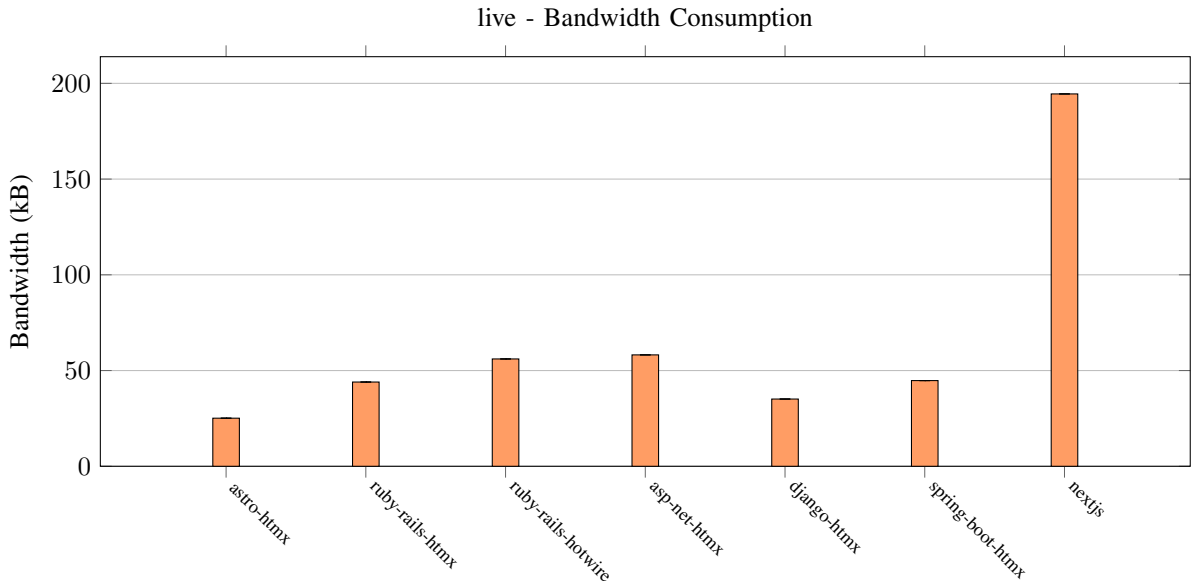


TABLE XXV: Statistical comparisons - `list` - Server energy.





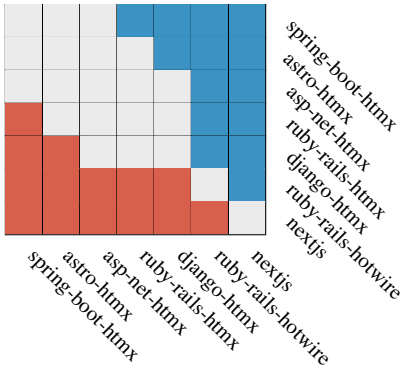


TABLE XXVI: Statistical comparisons - `live` - Client energy.

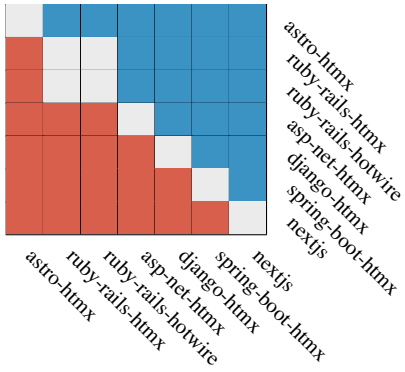
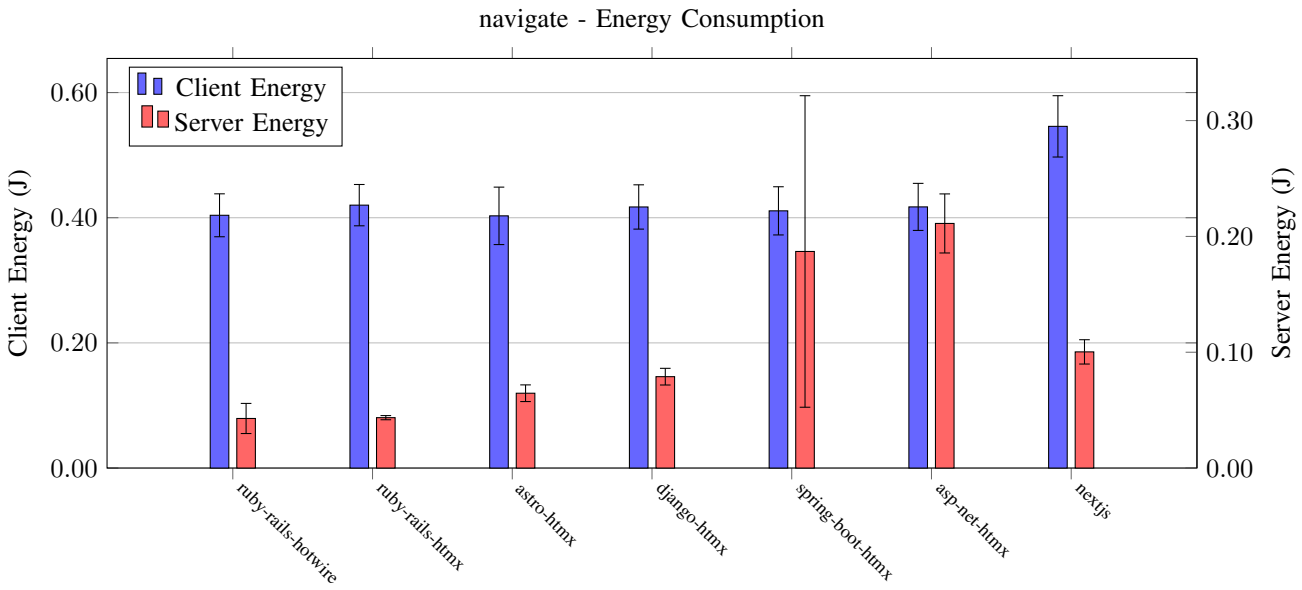
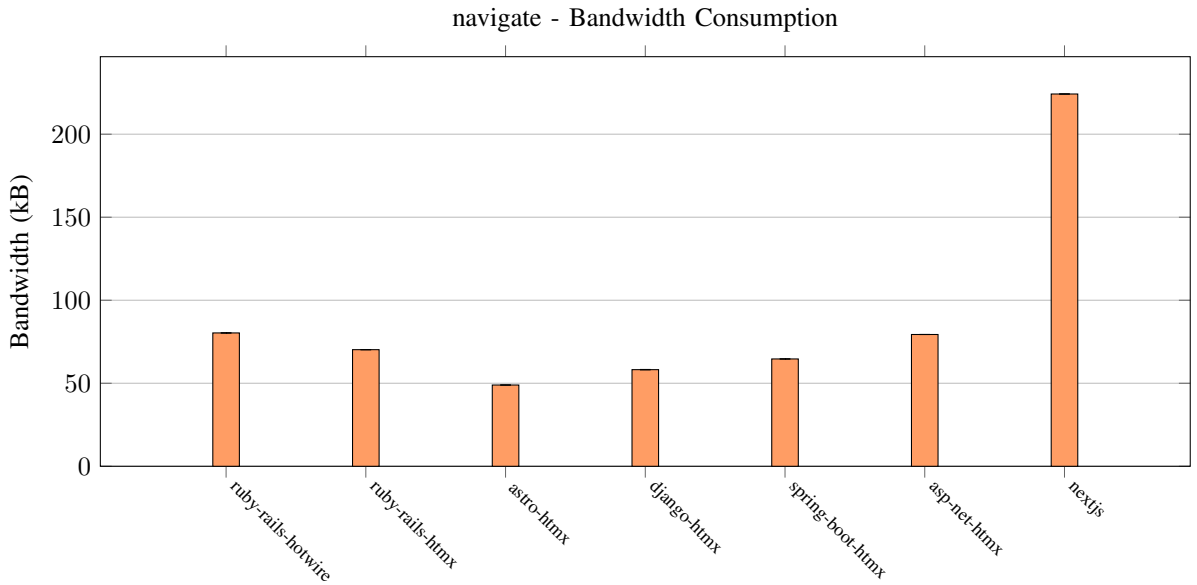


TABLE XXVII: Statistical comparisons - `live` - Server energy.





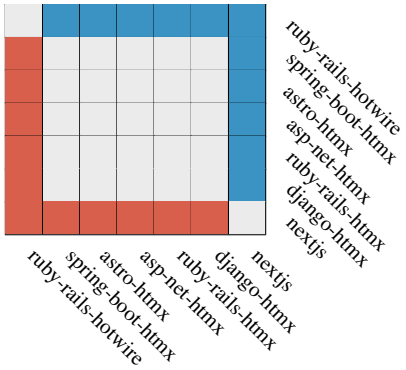


TABLE XXVIII: Statistical comparisons - `navigate` - Client energy.

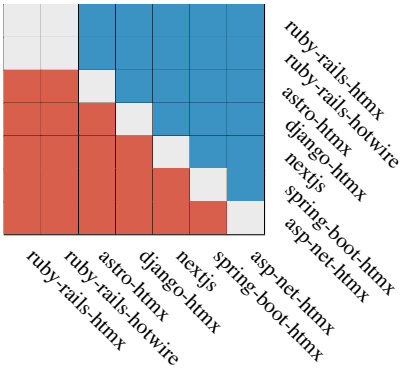
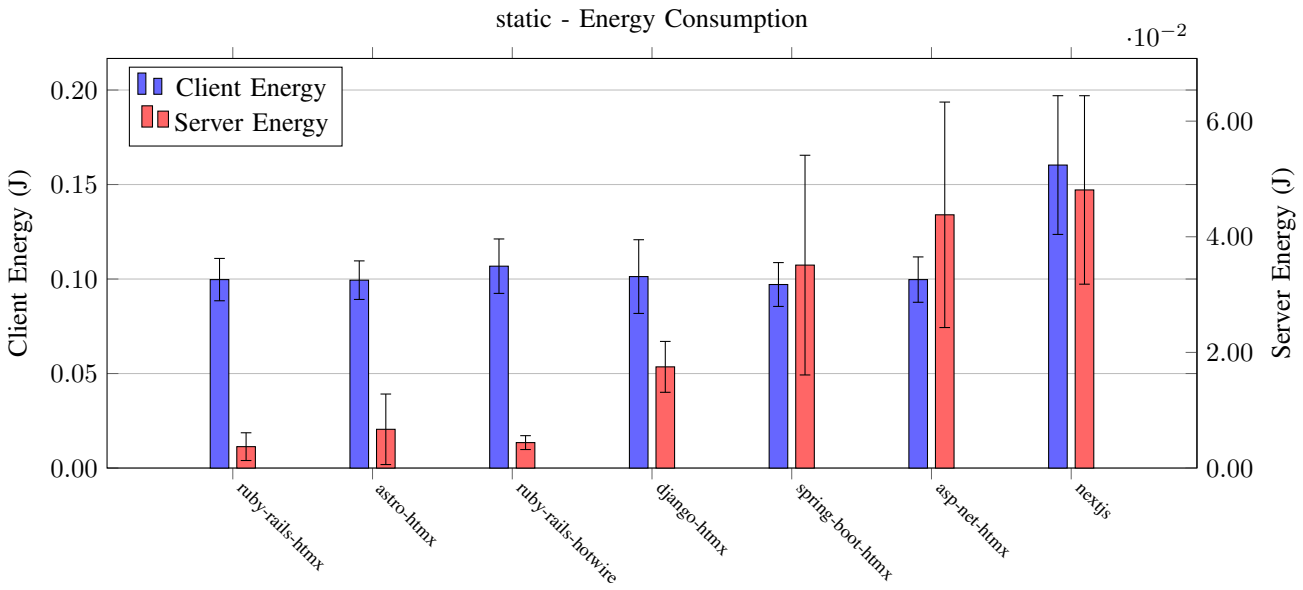
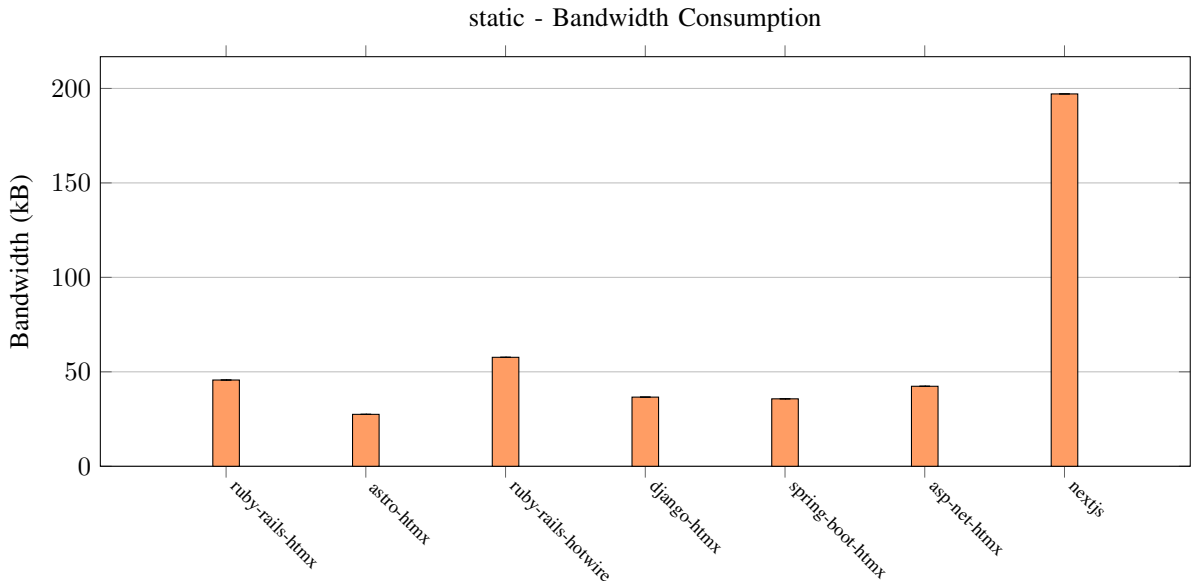


TABLE XXIX: Statistical comparisons - `navigate` - Server energy.





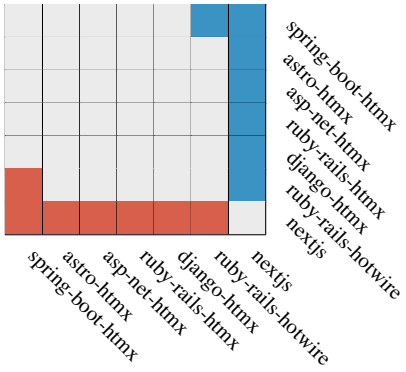


TABLE XXX: Statistical comparisons - `static` - Client energy.

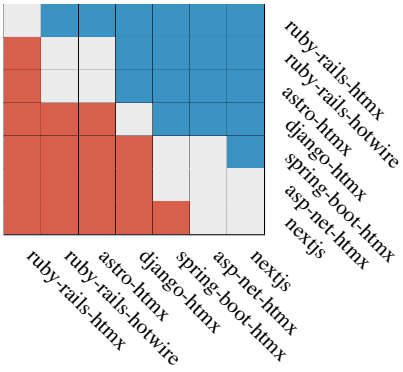


TABLE XXXI: Statistical comparisons - `static` - Server energy.

APPENDIX F: TRENDS OVER ITERATIONS

This section contains a chart for each test case that visualizes linear regressions across all iterations for each test site. The linear regressions allow us to determine whether the benchmark results exhibit systematic trends.

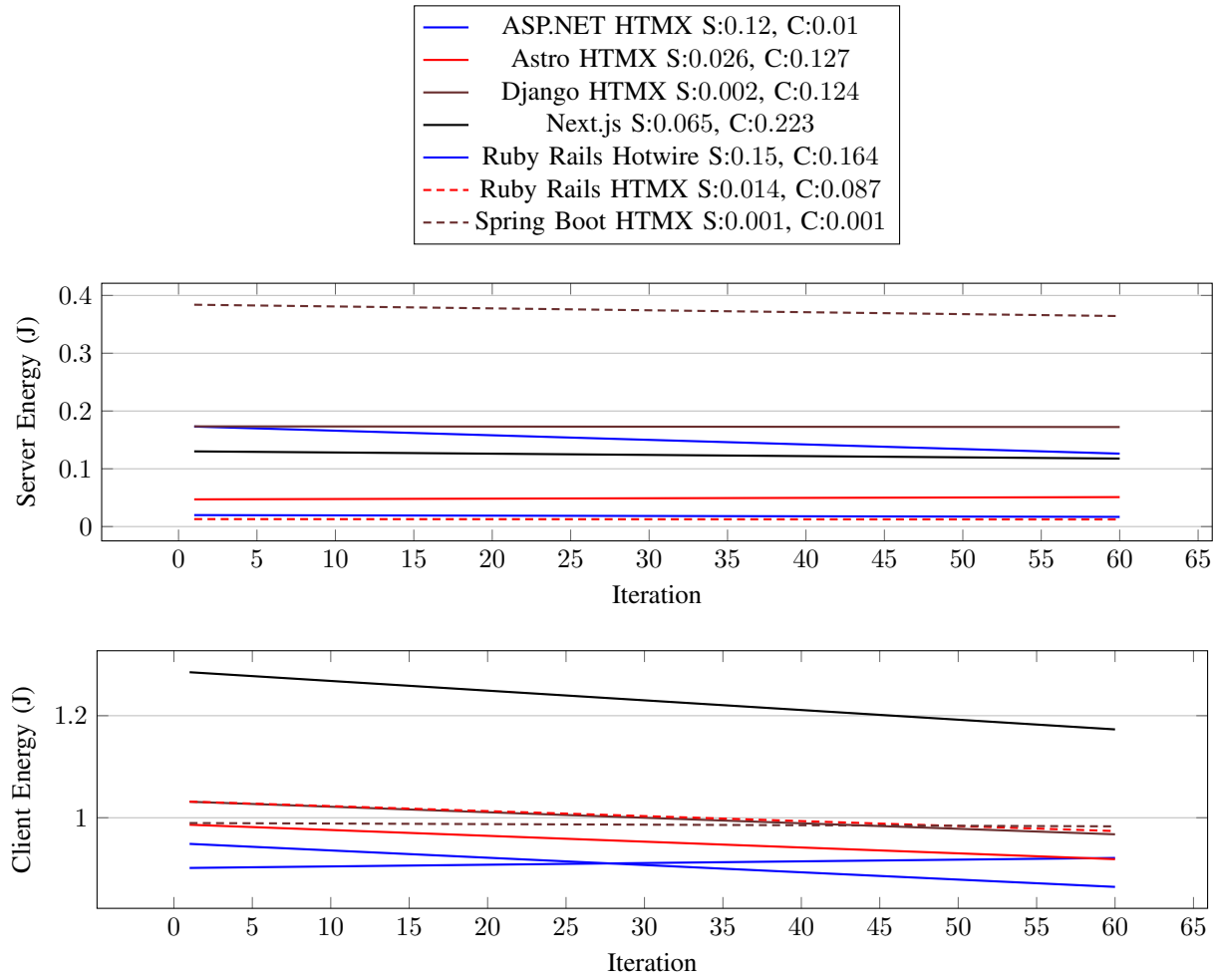


Fig. 21: home-interact - Linear regressions reflecting energy measurement trends. R^2 values in parentheses.

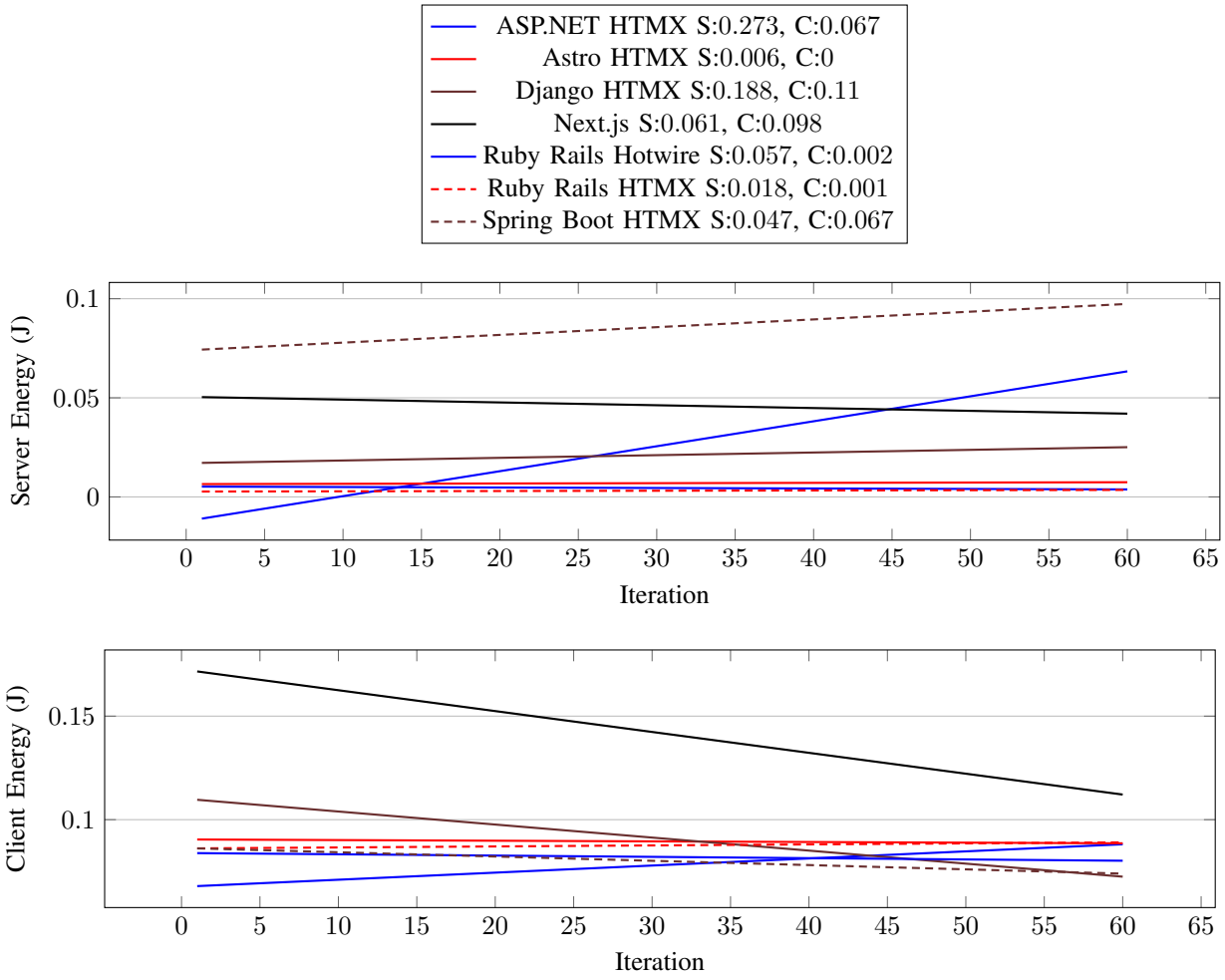


Fig. 22: list - Linear regressions reflecting energy measurement trends. R^2 values in parentheses.

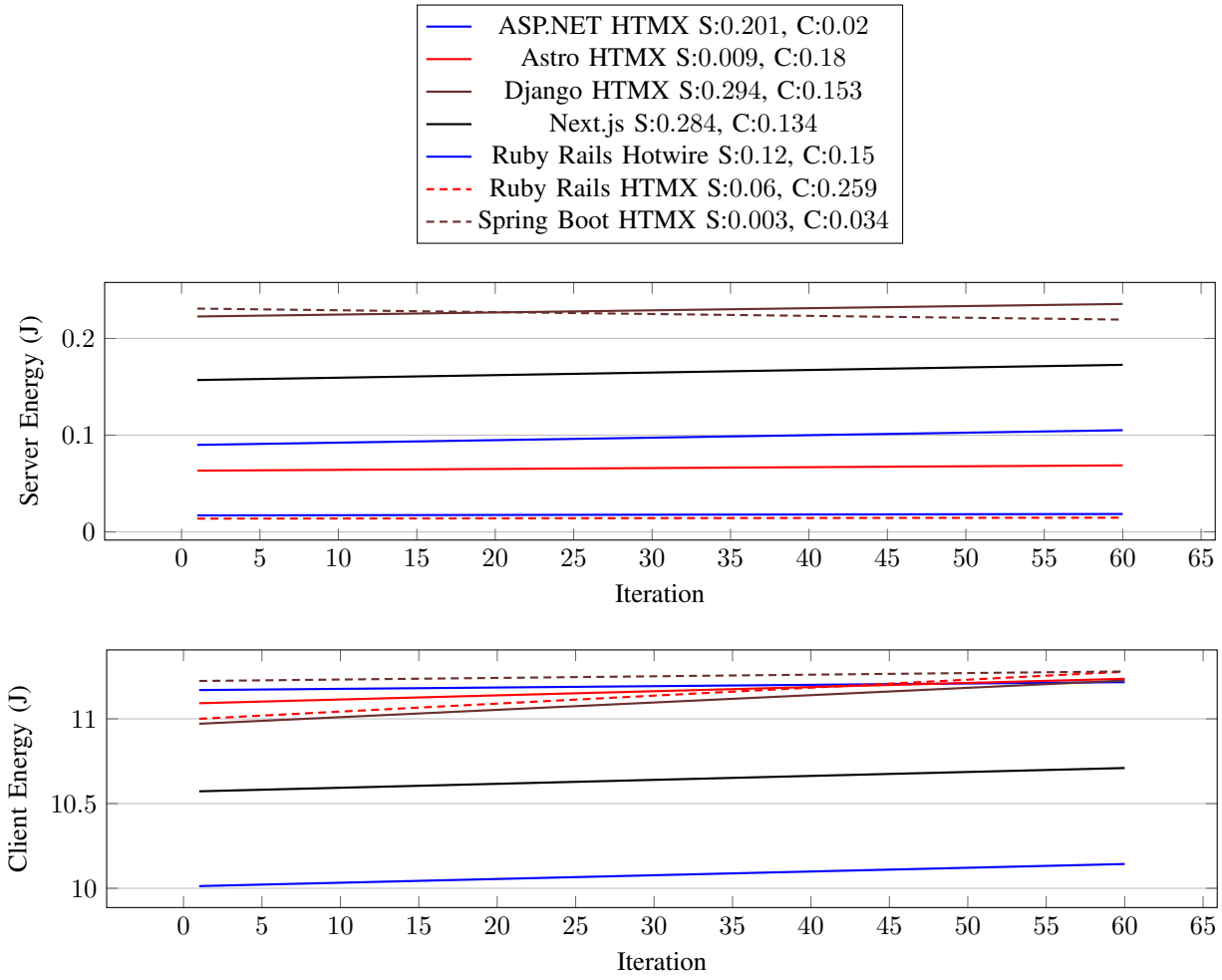


Fig. 23: list-interact - Linear regressions reflecting energy measurement trends. R^2 values in parenthesis.

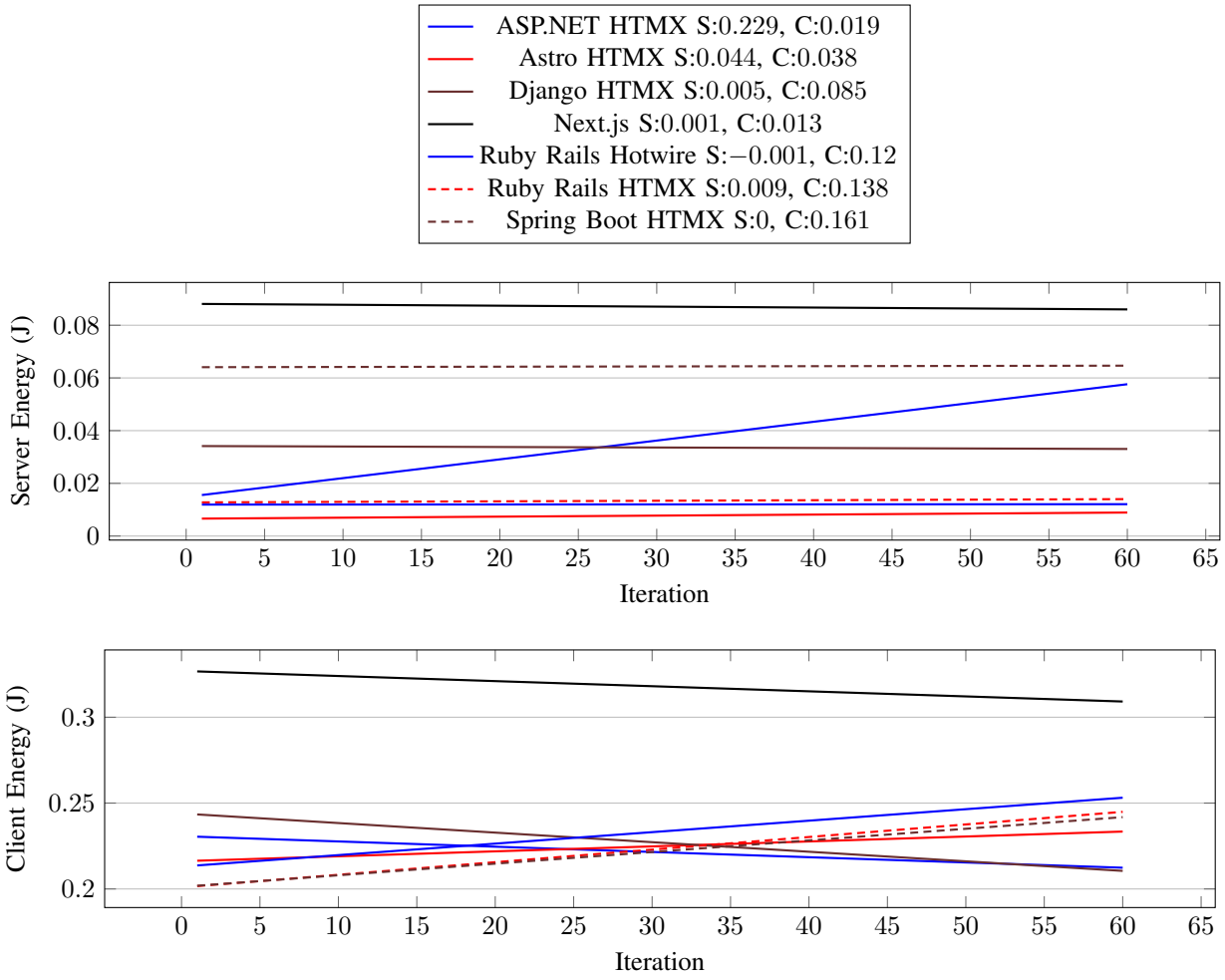


Fig. 24: live - Linear regressions reflecting energy measurement trends. R^2 values in parentheses.

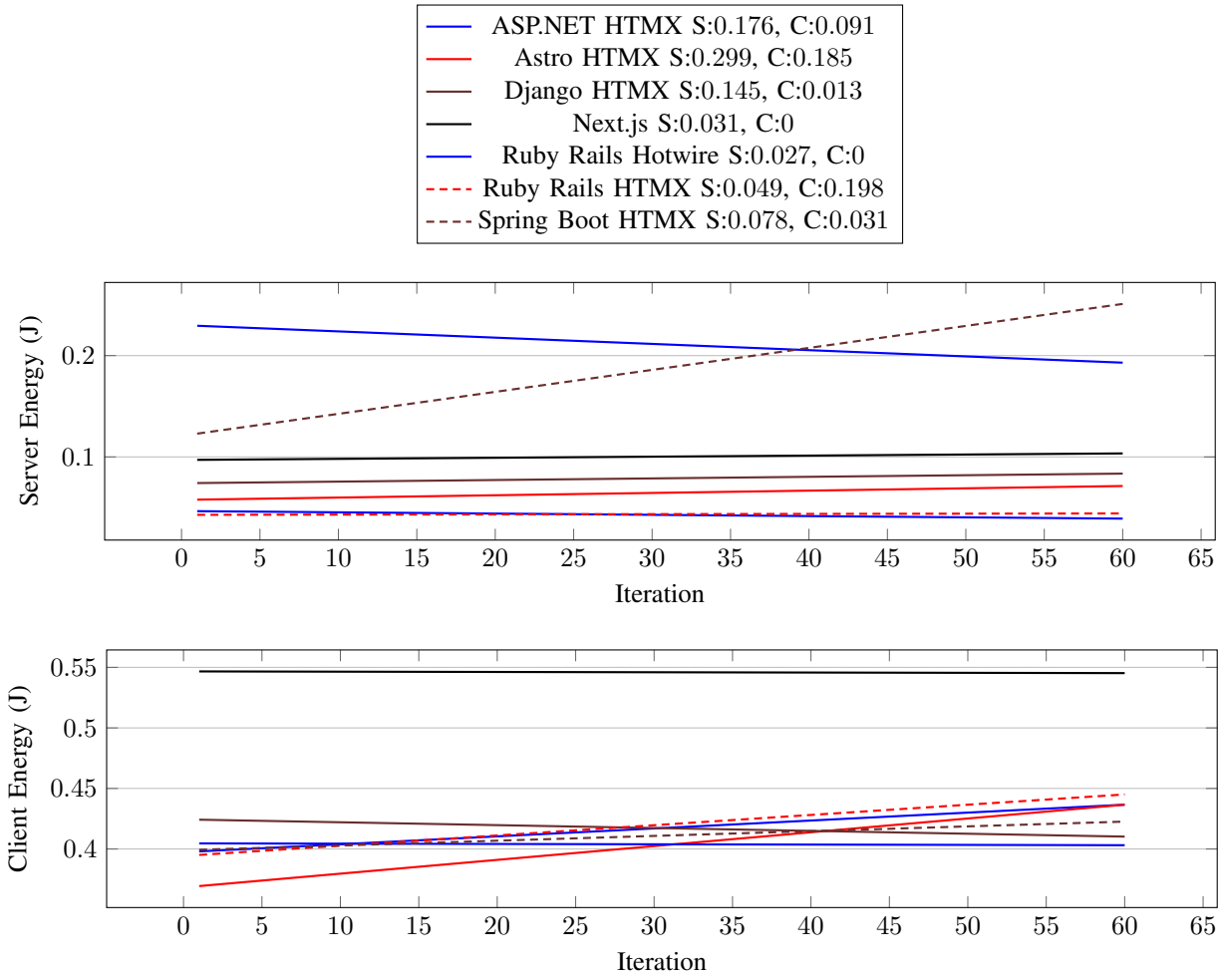


Fig. 25: navigate - Linear regressions reflecting energy measurement trends. R^2 values in parentheses.

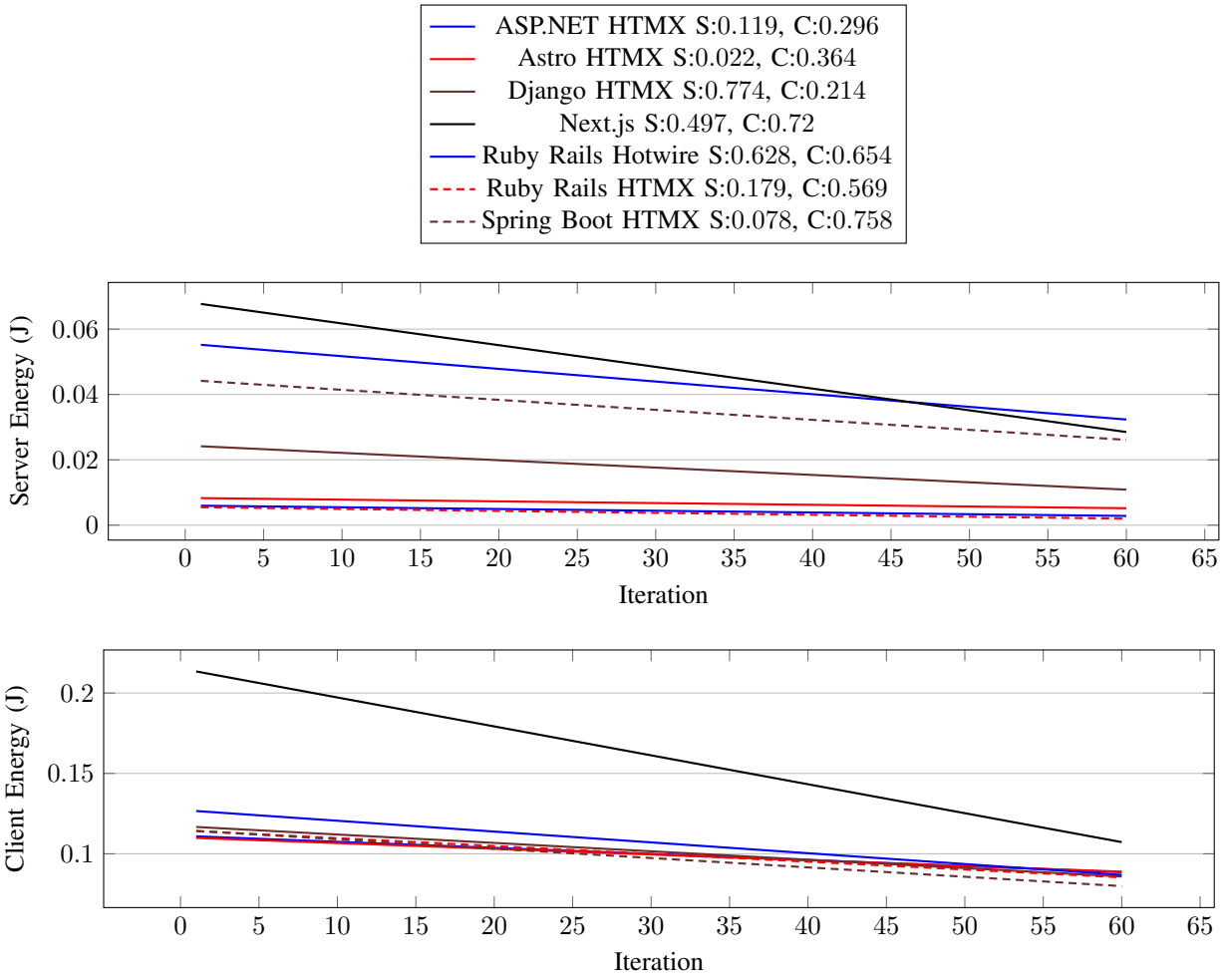


Fig. 26: static - Linear regressions reflecting energy measurement trends. R^2 values in parentheses.