Energy consumption of Node.js application frameworks for building RESTful APIs

Master Thesis

Roskilde University

Department of People and Technology

May, 2023
Supervisor: Maja Hanne Kirkeby (majaht@ruc.dk)
Co-supervisor: Kerstin Eder (cskie@bristol.ac.uk)

Authors:
Constantin-Razvan Tarau (constantin@ruc.dk, 72971)
Dragos-Andrei Mocanasu (dragosandr@ruc.dk, 72970)

Number of pages: 92
Number of characters: 162.158 (incl. spaces)

Table of contents

Acknowledgments

We would like to take this opportunity to express our profound gratitude and appreciation to the exceptional individuals who have made invaluable contributions to the realization of this thesis.

First and foremost, we extend our heartfelt thanks to our supervisors, Maja and Kerstin. Their unwavering support, insightful guidance, and extensive expertise have been instrumental in shaping the trajectory of our thesis. Their valuable input and mentorship have propelled us to new levels of achievement and nurtured our intellectual growth.

Additionally, we wish to express our sincere appreciation to the dedicated staff at FabLab, whose availability has been crucial to the success of our project. Among them, we would like to convey a special acknowledgment to Niels, whose expertise, assistance, and commitment to helping with the laboratory setup have played an indispensable role in ensuring the smooth and accurate execution of our measurements. We would also like to praise Luis Cruz for the cooperation provided by offering valuable guidance and advice during our research.

We are truly grateful for your support and for being an integral part of this transformative journey.

Table index

Figure index (excluding figures from the Appendix)

Abstract

This Master’s thesis presents a comparative analysis of the energy consumption of 4 of the most popular Node.js frameworks: Express.js, NestJS, Fastify, and Koa.js. In an era where energy efficiency is a critical factor in software development, understanding the energy profiles of different frameworks is crucial for optimizing resource usage and reducing environmental impact.

This study aims to measure and compare the energy consumption under a specific scenario of a RESTful API developed in the same way for all frameworks, using 2 different payload sizes for both the client and the server, which were executed on 2 Raspberry Pis. The measurements were external and done using a programmable power supply. Repeatability was ensured with the automation of all requests by a custom script.

The results of the study reveal notable differences in energy consumption among the examined frameworks. Fastify was the most energy efficient for a smaller payload, while Koa consumed the least with a larger payload. Express also demonstrates relatively low energy consumption due to its lightweight nature. NestJS exhibits the highest energy usage, potentially attributed to its extensive dependency injection and boilerplate code. The results have been validated through statistical analysis.

By understanding the energy consumption characteristics of these popular Node.js frameworks, developers can make informed decisions when choosing a framework that aligns with their functionality and energy efficiency goals, contributing to sustainable software development practices.

Glossary

- **AC** = Alternating Current
- **API** = Application Programming Interface
- **ARM** = Advanced RISC (Reduced Instruction Set Computer) Machine
- **BST** = British Summer Time
- **CDN** = Content Delivery Network
- **CEST** = Central European Summer Time
- **CLI** = Command-Line Interface
- **CMS** = Content Management System
- **CO2** = Carbon dioxide
- **CPU** = Central Processing Unit
- **CRUD** = Create, Read, Update, Delete
- **CSS** = Cascading Style Sheets
- **CSV** = Comma Separated Values
- **DC** = Direct Current
- **DOM** = Document Object Model
- **ECMAScript** = European Computer Manufacturers Association Script
- **Express** = Express.js
- **FP** = Functional Programming
- **FRP** = Functional Reactive Programming
- **GB** = GigaByte
- **GPIO** = General-Purpose Input/Output
- **GUI** = Graphical User Interface
- **HDMI** = High-Definition Multimedia Interface
- **HTML** = HyperText Markup Language
- **HTTP** = Hypertext Transfer Protocol
- **HTTPS** = Hypertext Transfer Protocol Secure
- **I/O** = Input-Output
- **ICT** = Information and Communications Technology
- **ID** = Identifier
- **IT** = Information Technology
- **JS** = JavaScript
- **JSON** = JavaScript Object Notation
- **KDE** = Kernel Density Estimate

- **Koa** = Koa.js
- **LCD** = Liquid Crystal Display
- **MB** = MegaByte
- **MSSQL** = Microsoft SQL Server
- **MVC** = Model-View-Controller
- **Nest** = NestJS
- **Node** = Node.js
- **OOP** = Object-Oriented Programming
- **OS** = Operating System
- **Ph.D.** = Doctor of Philosophy
- **RDBMS** = Relational Database Management System
- **REST** = REpresentational State Transfer
- **RP** = Raspberry Pi
- **SCPI** = Standard Commands for Programmable Instruments
- **SD** = Secure Digital
- **SDLC** = Software Development Life Cycle
- **SQL** = Structured Query Language
- **SSH** = Secure Socket Shell
- **TS** = TypeScript
- **UI** = User Interface
- **URI** = Uniform Resource Identifier
- **URL** = Uniform Resource Locator
- **USB** = Universal Serial Bus
- **USB-C** = Universal Serial Bus Type-C
- **XML** = Extensible Markup Language

1. Introduction

Keywords: Sustainable Software Development, Sustainable Software Engineering, Green Software, Green IT, API, Node.js, Energy Consumption, Energy Efficiency, Software Development.

Software Engineering is a discipline that involves designing, developing, implementing, testing, and maintaining software systems by applying validated engineering practices and principles through the use of tools, methodologies, and techniques to manage all software development processes. This is to ensure that high-quality software meets the needs of users and stakeholders while being generally reliable, scalable, performant, maintainable, and more efficient. Traditionally, this domain has focused on shortening the time-to-market and reducing costs (Bennett & Rajlich, 2000) while improving maintenance and encouraging changes (as seen for instance in the Agile Manifesto[1]). Unfortunately, areas such as sustainability and the long-term effects of energy consumption have been either ignored or left as an afterthought. Already, one approximation according to (Belkhir & Elmeligi, 2018) is that the Information and Communications Technology sector could contribute to more than 14% of the global carbon footprint by the year 2040. Although more recent estimates (Andrae, 2020) argue that there is a probability of the CO2 emissions from the digital sector to decrease between 2020 and 2030, they are based on scenarios where the adoption of renewable energy sources is higher than the current rate.

Sustainable Software Engineering can be defined as a field that, similarly to Software Engineering, is concerned with the creation of software, however in this case, of systems that are not only functional and reliable, but also environmentally, socially, and economically sustainable. In other words, such systems are designed and developed to minimize their environmental impact, promoting social responsibility while ensuring long-term economic viability. In essence, this field encompasses multiple interconnected dimensions (Becker et al., 2015), such as technical, economical, environmental, social, and individual.

The research domain of Sustainable Software started to gain popularity in recent years (Wolfram et al., 2017) (Calero et al., 2020), even though just a few countries have been contributing in the area. When looking at the literature, we find articles (Naumann et al., 2011) (Penzenstadler, 2015) (Pereira et al., 2017) (Kern et al., 2018) that talk about how each phase of the software development process (Singh, 1996) should be given special consideration, such that the sustainability impact is assessed and evaluated through certain measures and checklists, using specialized tools, incorporating sustainable methodologies, and by constantly reevaluating how work is performed. Some of these measures could entail changes in the distribution, packaging, and recycling of technology products, implementation of design patterns in code, deployment mediums, decommissioning, and disposal, virtualization strategies, and specific guidelines for different roles (for instance split for users, developers, and administrators).

[1] Principles behind the Agile Manifesto: https://agilemanifesto.org/principles.html

While in practice the impact would result from embodying sustainability in all software development lifecycle (SDLC) phases, for this thesis, we have decided to narrow the scope and look at one specific area from Green Software, namely software development (i.e. the technical dimension of Sustainable Software Engineering). The topic we have chosen is to investigate the possibility of measuring and comparing the energy consumption of different Node.js application frameworks used for the same purpose, that of building APIs that follow the RESTful architectural style[2].

Energy transparency is defined as providing energy consumption to the developer, which holds great potential to empower software engineers in making more informed decisions, as well as facilitating and encouraging the development of specialized tools aimed at selecting the most efficient code (e.g. by analyzing code snippets or algorithms and providing recommendations). This transparency goes beyond merely understanding resource consumption and extends to capturing and visualizing energy usage within software systems. By making energy consumption visible, engineers can gain valuable insights into the efficiency and sustainability of their code, leading to better decision-making and resource optimization. With this visibility, they can identify energy-intensive components and pinpoint areas where energy efficiency can be improved. By understanding energy consumption patterns, engineers can make informed choices about code optimization, algorithm selection, and system configuration (Eder et al., 2016). Through this notion, our objective is to provide developers with additional insights and information. By exploring the energy transparency of Node.js frameworks in our thesis, we strive to enhance developers' understanding of the energy implications associated with their code and empower them to make more informed decisions regarding energy-efficient programming practices.

1.1. Thesis Goal

Thus, the overall goal is to investigate the energy efficiency of the top most popular Node.js frameworks. To measure the energy consumption, we created a setup that simulated a scenario from production-ready systems, by hosting a large, public database online and making external calls through the network, coding the APIs to contain request validations for each endpoint, all while trying to remove any possible factor that could cause fluctuations and make it more to compare their measured energy efficiency.

1.2. Proposed research questions

1. How can we measure the energy consumption of Node.js frameworks?

2. Which of the most popular Node.js frameworks for developing APIs are the most energy efficient?

[2] What is REST: https://restfulapi.net/

1.3. Audience

The findings of our thesis can benefit multiple audiences and generally brings awareness among the public interested in Green Software.

The main topic is Node.js application frameworks used for creating RESTful APIs and their energy usage, thus we consider the main audience to be composed of software engineers who are concerned with developing more sustainable and efficient software, either for internal or external use. Our results could empower developers with energy-efficiency information, allowing them to consider energy consumption as a first-order design goal. In return, this creates awareness among vendors and suppliers and encourages them to focus on changing current or creating new lines of more sustainable products and services for the world of Information and Communication Technologies.

2. Thesis Methodology

This chapter describes the main processes, practices, and principles we have used during our thesis period.

In order to answer our research question, we needed to conduct an experiment where we compare Node.js frameworks in the same environment, performing the same tasks under the same conditions. The experiment can be labeled as explorative, due to the frameworks being measured and compared for their energy consumption, something that we were not able to find anywhere else during our literature review. The only similar experiment for the same frameworks that we were able to find was in (Demashov & Gosudarev, 2019) yet the authors were assessing performance and latency, and not the energy consumption. The outcome and the way the experiment is conducted will determine the thesis conclusion and the answers to the research questions.

The integration of theory in the initial stages of the thesis is imperative for establishing a base to start from. The use of literature serves multiple objectives such as consolidating the research domain as well as developing a comprehensive repository of fundamental principles related to energy measurements in software development. Furthermore, literature is employed to evaluate the boundaries of the experiments and to present the limits that naturally arose as a result of temporal, financial, and architectural constraints. The limitations and the resulting outcomes of the thesis are presented in the “7. Discussion” chapter.

Our thesis process was inspired by practices from Agile Project Management (i.e. SCRUM) and from our daily work as software engineers. By this, we mean that all tasks for the next weeks were prioritized. Additionally, when looking at the previous week (i.e. during our retrospective meetings) we asked ourselves what has been achieved and what could have been improved. If we decided to split some tasks to increase productivity, the following day started with a stand-up meeting (usually in person) where we briefed each other on the changes, the challenges and discussed the proposed solutions for the next milestones. For current tasks and subtasks, we have incorporated a Kanban board[3] where we sorted the tasks based on their status (i.e. “To Do”, “Doing”, “Done”). All future tasks were stored in a backlog. In essence, each month that was part of our thesis process contained established objectives and goals (epics), based on which we created the main tasks (stories), and then actionable subtasks.

Collaboratively, we have worked and shared resources amongst each other by using Google Drive[4] due to the platform’s ease of file distribution and the ability to write the same document and at the same time, the free plan (on the basis that each collaborator possesses a Google account), and integration with external extensions, or so-called “Add-ons” in Google’s terminology. One such add-on was our preferred tool for managing citations and references: MyBib[5] is a free

[3] What is a Kanban board: Atlassian
[4] Google Drive: https://www.google.com/drive/
[5] MyBib: Chrome Web Store

extension found on the Google Web Store that can search for books or journal articles, keep track of all sources, and integrate with Google Docs or Microsoft Word for in-text citations.

We have also benefited from weekly or bi-weekly meetings with our supervisors. Before each meeting, we ensured that all we have worked on the past week was captured in an email sent before the supervision sessions, allowing us to keep the focus on current problems, discuss the proposed solutions, and optimize the time. These sessions were held either on-site at Roskilde University or online via Zoom.

Finally, we also documented the high-level objectives in a logbook (see “10.1. Logbook” in the Appendix), where we wrote the goal that we were trying to achieve in a week or month, together with a more detailed description of that particular goal.

3. Background

This chapter briefly describes the context of each fundamental technology and concept that is part of our experimental setup. Later in chapter "4. Experiment Design", we will explain the details of each subcomponent (including both hardware and software) that was part of the more general or high-level technologies presented below.

3.1. JavaScript

JavaScript[6] is a high-level, dynamic, and untyped interpreted programming language that was initially used to create interactive web applications, allowing interactivity between the user and the elements from the web page. JavaScript can be used to manipulate the contents of the web page (i.e. the DOM), offering users the possibility of dynamically adding or removing elements on the application (Flanagan & Novak, 1998). JavaScript is usually implemented together with HTML, which is used to bring structure to the information on a website, and CSS, which is used to make the web page more attractive. They represent the foundation of frontend web development (Jennifer Niederst Robbins, 2018).

JavaScript evolved until its current 13th ECMAScript version (at the time of writing this thesis) and is being further developed in different areas of web development. Frontend frameworks such as React, Angular, and Vue were developed based on JavaScript and are now used to build frontend applications, creating responsive web and mobile software (Persson, 2020). Moreover, Microsoft developed TypeScript (TS) which is a superset of JavaScript, with the addition of being strongly typed, like other popular programming languages such as Java, C, or C#. Being strongly typed, TS eliminates one of the main JavaScript disadvantages by making it less prone to logic errors.

Recently, JavaScript became a popular programming language for the server side too, with the constant development of Node.js as a backend environment based on JavaScript. We decided to use JavaScript with Node.js to build RESTful APIs in multiple Node.js frameworks to compare their energy efficiency, due to the overall popularity, flexibility, and ease of programming. We also took into consideration our previous professional and academic experience when deciding on JavaScript as the programming language for this research. According to the StackOverflow survey[7] conducted in 2022, JavaScript has once again secured its position as the most commonly used programming language for the tenth consecutive year. This achievement underscores the enduring popularity and widespread adoption of JavaScript among developers worldwide. With its continued dominance in the programming landscape, JavaScript reaffirms its status as a foundational language for creating cutting-edge web applications and driving innovation in the digital realm.

[6] JavaScript: https://www.javascript.com/
[7] Link to the survey: https://survey.stackoverflow.co/2022/

3.2. Node.js

Node.js[8] is an open-source, cross-platform, and event-driven JavaScript environment. It offers an asynchronous, event-driven design and is based on the JavaScript engine found in Chrome V8. Node.js was developed by Ryan Dahl in 2009, and since then, it has grown extremely popular among developers for creating server-side programs, APIs, and web applications. Developers can now use JavaScript to create client-side code which is executed in web browsers, and server-side code thanks to Node.js. Furthermore, Node.js can also be used to create web-based applications and APIs using a variety of integrated modules and tools, including the file system, HTTP, and HTTPS (Satheesh et al., 2015).

One of Node.js's main advantages is its non-blocking I/O model, which enables it to manage high volumes of traffic without encountering any delay. Another benefit of Node.js is its sizable and active developer community, which has aided in the development of a thriving ecosystem of third-party tools and packages (Satheesh et al., 2015).

Among the reasons that make Node.js a popular choice when developing web-based or server-side applications and APIs (Bangare et al., 2016) we can name:

1. Being based on Google Chrome’s JavaScript V8 engine, which makes it fast and efficient as the engine is written in C++.

2. Easy to use and implement: in just 5 lines of JavaScript code, one can have a web server running as shown below:

```
var http = require('http');
http.createServer(function (req, res) {
  res.write('Hi from server');
  res.end();
}).listen(3000);
```

3. Large ecosystem of frameworks and libraries: many engineers support the community and write frameworks built on top of Node to speed up and streamline development.

4. Asynchronous behavior: Node can handle numerous requests on a single thread without being slowed down by additional I/O operations. To accomplish this, all requests are handled concurrently by an event loop.

[8] Node.js: https://nodejs.org/en/

3.3. RESTful APIs

API is an acronym that stands for Application Programming Interface and it represents a "contract" between an information consumer and a provider of information. Between the client and the server, an API acts as a mediator so that clients can tell the system what they require, and the system can comprehend and execute those requests (Satheesh et al., 2015).

Roy Fielding defined the architectural style known as REST, or Representational State Transfer, as part of his Ph.D. dissertation, in which he specified a number of standards for developing web services (Fielding, 2000) (Masse, 2011). The set of REST principles defined by Fielding are the following:

1. **Client-server architecture** refers to the separations of concerns between server and client. The client is the application that makes a request, and the server is the application that takes the request, processes it, and sends a response back to the client. They can be implemented independently and are technology agnostic.

2. **Uniform interface**, which consists of 4 main components:
    a. Identification of resources in requests:
        i. RESTful APIs use URIs (Uniform Resource Identifiers) to identify and locate resources. Clients make requests to specific URIs to perform operations on those resources, such as retrieving, creating, updating, or deleting them.

    b. Resource manipulation through representations:
        i. Resources are represented in various formats, such as JSON or XML. Clients interact with resources by sending representations of the resource in requests and receiving representations in responses. These representations encapsulate the state and data of the resources being manipulated.

    c. Self-descriptive messages:
        i. RESTful messages contain all the necessary information for the recipient to understand and process them. The messages include metadata, such as headers, that describe the content, format, and caching instructions. This self-descriptiveness allows clients and servers to communicate effectively without prior knowledge or assumptions.

    d. Hypermedia as the engine of application state (HATEOAS):
        i. It means that the API responses contain hyperlinks or links that guide clients on how to navigate and interact with the available resources. By following these links, clients can dynamically discover and transition to different application states, enabling a more flexible and decoupled system.

3. A **layered system** allows network-based intermediaries to exist and monitor client-server interactions. As a result, it maintains the client and server separate, enabling other servers to provide load balancing, security, and caching.

4. **Caching** is one of the most important constraints, instructing the web server to declare the cacheability of each response's data, to improve performance and reduce network traffic.

5. **Statelessness** suggests a situation where the server does not know the client's current state. Each request submitted by the client contains all the relevant information that is necessary for communicating with the server.

6. **Code-on-demand** is an optional constraint that enables web servers to temporarily transfer executable programs, such as scripts or plug-ins, to clients.

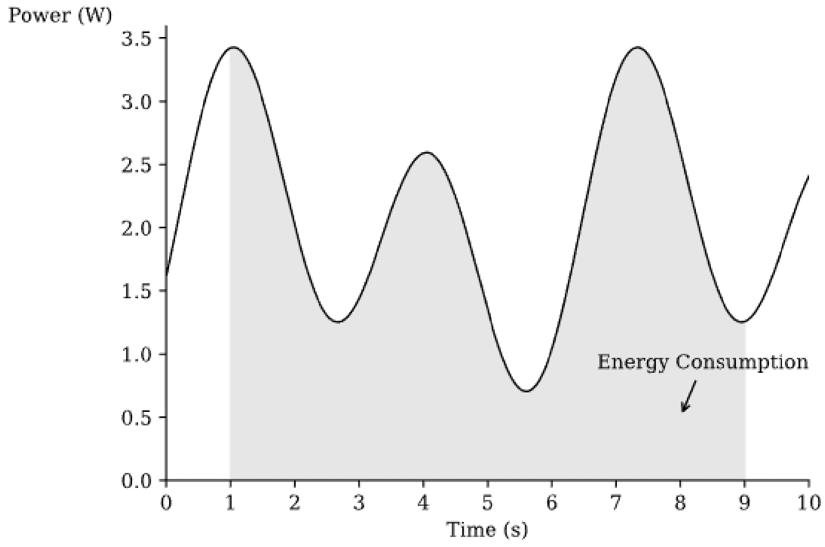
RESTful APIs are a form of web service that enables data or features to be accessed over the internet using standardized HTTP methods. RESTful APIs use URLs to designate resources and HTTP methods to specify what clients want to do with those resources. GET, POST, PUT, and DELETE are the HTTP methods that are most frequently used in RESTful APIs. (Satheesh et al., 2015).

3.4. Electric power

Understanding the concept of electric power in the context of sustainable software is crucial as it allows developers to optimize energy consumption, reduce environmental impact, improve cost efficiency, promote green computing practices, and comply with regulatory requirements. By comprehending how electric power is utilized within software systems, professionals can design energy-efficient solutions, minimize operational costs, and contribute to a more sustainable and responsible digital ecosystem.

Electric power is the rate at which energy is transferred, transformed, or consumed per unit of time. It is an instantaneous measure, meaning it gives a measure of the amount of energy used or produced at a particular moment in time. It is measured in Watts (W) and is calculated by multiplying the voltage (V) measured in Volts by the current (I) measured in Amperes, thus having the formula $P = V * I$. However, to understand the total energy used or produced, it is necessary to measure power over time resulting in a curve that shows the power as a function of time. This curve is often called a power-time graph or power curve. For instance, a power-time graph can be used to analyze the energy consumption of a system over the course of a day. By measuring the power usage at different times and plotting it on a graph, one can see the times when the power usage is highest, indicating periods of high energy consumption. (Ling et al., 2016)

Related to power, the area under a curve is the total amount of energy consumed over a certain period, represented by a graph of energy consumption as a function of time. The area is calculated by integrating the energy consumption function over the specified time interval. For example, if the power (W) is plotted on the y-axis and time (s) is plotted on the x-axis, the area under the curve between 2 specific times represents the amount of energy consumed during that time. Both the power as a function over time and the area under the curve can be seen below in Figure 1.



,

Figure 1. Energy consumption over time is seen in the area under the curve. Image source: Luis Cruz in EnergyMetrics.ipynb - Colaboratory (google.com).

3.5. Energy

Electric energy refers to the capacity of an electrical system to do work. The unit of electric energy is the watt-hour (Wh) or kilowatt-hour (kWh), which represents the amount of energy consumed or generated in one hour at a constant rate of power. For instance, a 100-watt light bulb that is turned on for 10 hours consumes 1 kilowatt-hour of electric energy.

The formula for electric energy is $E = P * t$, where E is energy in watt-hours (Wh) or kilowatt-hours (kWh), P is power in watts (W) or kilowatts (kW), and t is time in hours (h). This formula expresses the relationship between energy, power, and time in an electrical circuit. The result is expressed in Joules (J) and it is used to calculate the amount of energy consumed by that circuit over a given time. On a graph, this can be seen in the area under the curve. (Ling et al., 2016)

3.6. Methods to approximate the energy consumption

To calculate the total energy consumption, one can approximate the area under the curve by using the trapezoid method, a numerical integration technique that works by dividing the area under the curve into a series of trapezoids and calculating the sum of their areas to approximate the total area. Specifically, the interval of integration is divided into several subintervals, and the function is evaluated at the endpoints of each subinterval. The area of each trapezoid is then calculated as the average of the function values at the endpoints multiplied by the width of the subinterval. The sum of the areas of all the trapezoids gives an estimate of the area under the curve. The trapezoid method is used to estimate energy consumption because it provides a more accurate approximation of the energy consumed by devices with varying power draw. Unlike other methods that assume a constant power draw or use the average power consumption, the trapezoid method takes into account the changes in power consumption over time. This method is particularly useful for devices that exhibit dynamic power behavior, such as those that have variable workloads.

Another approach to estimating energy consumption involves multiplying the average power of a system by the duration of a task execution. The formula is $W = Pavg * \Delta t$, where W is the consumption, Pavg is the average power consumption, and Δt is the duration of the activity. This method is commonly employed in cases where the power consumption remains constant throughout the entire duration, as opposed to the trapezoid method.

3.7. Power supplies

A power supply is an electronic device that converts the electrical power from a source, such as a wall outlet, into the correct form and voltage needed to power another electronic device or system. It works by taking in the alternating current (AC) from the power source, which is usually at a high voltage and frequency, and transforming it into a direct current (DC) at a lower voltage and frequency that is suitable for use by specific electronic devices. The power supply contains several components, among others being a transformer, rectifier, and voltage regulator. The transformer is used to step down the high-voltage AC power from the wall outlet to lower-voltage AC power, which is then passed through a rectifier that converts the AC power into DC power. The voltage regulator then controls the output voltage to ensure that it remains constant and within a specific range, even if the input voltage or load on the power supply varies. (Brown, 2001)

Power supplies are designed to be efficient, meaning that they convert as much of the input power as possible to the output power without dissipating it as heat. Efficiency is typically expressed as a percentage and is calculated by dividing the output power by the input power. For example, a power supply that has an efficiency rating of 75% would convert 75% of the input power to output power and dissipate the remaining 25% as heat.

Several factors can affect the efficiency of a power supply. The quality, design, and technology used, the surrounding environment (including temperature and humidity), and the input and output power can significantly impact the efficiency of a power supply. Thus, the efficiency of a power supply is not constant and can vary depending on the load. Most modern power supplies (at least the ones holding an 80 PLUS certification[9]) have efficiencies in the range of 80-90%, with some high-end models exceeding 90%. Lower-end or older power supplies may have

[9] Efficiency table: https://www.clearesult.com/80plus/program-details#program-details-table

efficiencies in the range of 50-70%. At lower loads, the efficiency drops off, while at higher loads, the power supply can become less efficient due to heat dissipation. In Figure 2 we can see an example of how the efficiencies differ between 2 power supplies with different voltages due to their maximum output power (one with 115V and the other with 230V).

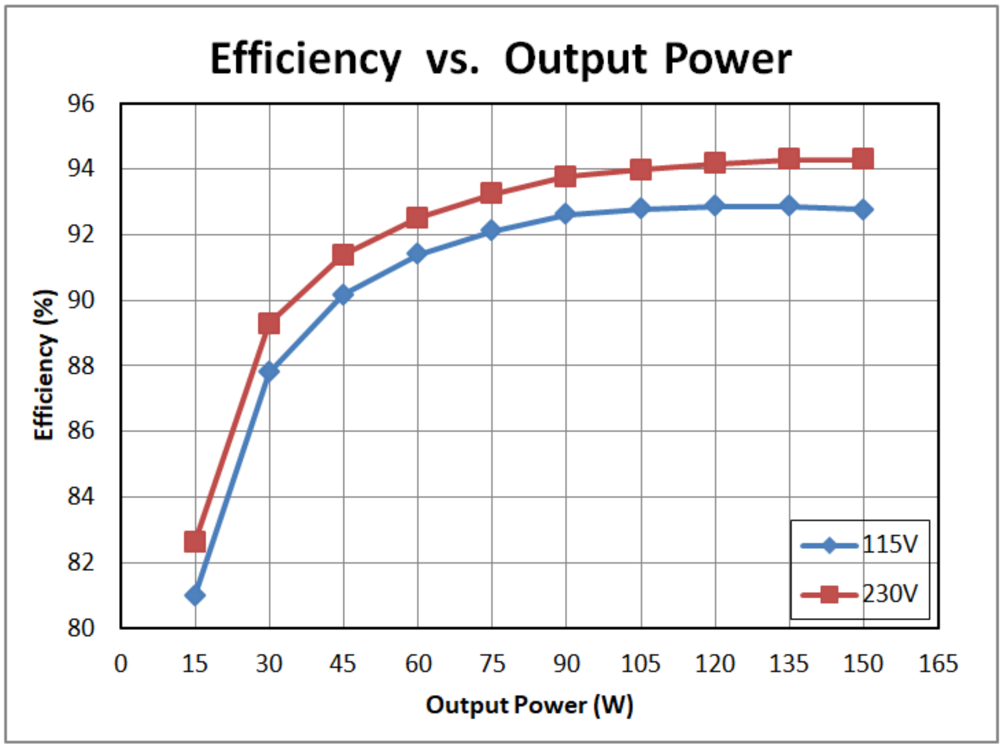


Figure 2. Efficiency curves of 2 power supplies with different voltages.
Image source: https://e2e.ti.com/.

Therefore, it is crucial to measure and control external factors, while assessing all qualities of a power supply before using one to power electric devices. In the next chapter called “4. Experiment Design”, we will discuss our programmable power supply of choice, which was also integrated with a piece of software written in Java that allowed us to extract data related to power and energy consumption when performing API calls.

3.8. CPU temperature

CPU temperatures can have a significant impact on the processing power of a computer. As a CPU operates, it generates heat, and if that heat is not properly dissipated, the CPU can overheat and suffer damage. To prevent this, most CPUs have a built-in thermal protection mechanism that reduces the processing power when the temperature reaches a certain level.

The thermal protection mechanism works by throttling the CPU, reducing its clock speed and voltage. This diminishes the processing power of the CPU, as it takes longer to execute instructions at a lower clock speed. As a result, if a CPU is constantly running at a high temperature, it may not be able to perform at its maximum processing power. What is more, high temperatures can cause instability in the CPU, leading to crashes, system freezes, or even permanent damage. The maximum safe temperature for a CPU varies depending on the model, but most popular CPUs have a maximum safe temperature of around 100-105 degrees Celsius[10].

Therefore, to ensure that a CPU is operating at its maximum processing power, it is essential to keep its temperature within a safe range. This can be achieved by using proper cooling solutions, such as fans or liquid cooling, and by ensuring that the CPU is not overclocked beyond its safe limits. Additionally, proper airflow and ventilation in the computer case can help dissipate heat more effectively. Furthermore, when the CPU is used continuously for long periods, it can cause the temperature to rise even further, which can lead to a decrease in performance or even damage to the CPU. To avoid this, the CPU needs to be given time to cool down. This can be achieved by shutting down the system or by allowing the CPU to idle for a while until it reaches a lower temperature. In our case, we set the CPU to rest for a minute after each execution of 30 calls, together with a 5-minute baseline wait before and after each experiment. More details about the experiment methodology are found in the chapter “4. Experiment Design”.

When performing experiments for energy consumption, it is important to consider the impact of the CPU temperature, as this affects power consumption in 2 ways: first, as temperature increases, the power consumption increases too; second, as temperature increases, thermal throttling may occur, which reduces the CPU's processing power to prevent damage, thus compromising performance and the results. Similarly, it is important to allow the CPU to cool down after the measurement process is complete. This is to make sure that any residual heat does not influence subsequent measurements.

3.9. Python and Jupyter Notebook

Python is a high-level, interpreted programming language that is often used for data analysis, statistics, and scientific computing. It is easy to learn as it has a simple syntax and a large community of users who contribute to a wide range of libraries and packages.

We have decided to use Python because it is a tool known for its utility for statistical analysis and graphing due to the availability of powerful libraries such as NumPy, Pandas, SciPy, and Matplotlib. NumPy provides fast and efficient numerical operations, while Pandas is a library for data manipulation and analysis. Matplotlib is a plotting library that allows you to create high-quality visualizations of your data, while Seaborn is a data visualization library based on Matplotlib, which provides a higher-level interface for creating statistical graphics, such as bar plots, heat maps, scatter plots, or line plots (Mckinney, 2013).

A popular tool for working with Python is Jupyter Notebook[11], a web application with computation documents where text and figures can be inserted, together with live code that can be executed

[10] More information on Intel processors and their maximum temperature: https://www.intel.com/content/www/us/en/support/articles/000005597/
[11] Jupyter Notebook: https://jupyter.org/

from browser-based cells, without the need for a separate development environment. This interactivity between text, code, and data visualization facilitates exploration and the presentation of data from research (Perkel, 2018). Other notable features include collaboration since other users can easily read, edit and add comments, reproducibility since notebooks can be used to document an analysis workflow and then shared with others for their testing, and versatility, supporting more than 40 programming languages. Interestingly, the name Jupyter is derived from the 3 main core programming languages it supports, namely Julia, Python, and R.

3.10. Statistical analysis

The purpose of this chapter is to delineate the statistical tools employed in data validation and analysis for comparing measurement data samples. These statistical methods are also utilized and mentioned in the “5. Data Preparation and Analysis” segment of this thesis.

3.10.1. Definitions and terminology

The goal of statistical analysis is to make sense of the data collected and to help draw meaningful conclusions from that data. To accomplish this, there are several key concepts and terms that must be understood.

One of the most common measures used in statistical analysis is the mean, also known as the average. It is calculated by taking the sum of all values in a dataset and dividing it by the total number of values. The mean is a useful measure of central tendency that helps us understand the typical value of a dataset.

Another important concept is the standard deviation. It measures the amount of variation or dispersion in a dataset, indicating how far the values in a dataset are from the mean. A high standard deviation means that the values in the dataset are spread out, while a low standard deviation means that the values are tightly clustered around the mean.

Finally, hypothesis testing is a statistical technique used to determine whether an observed effect or relationship in a sample is statistically significant and likely to have occurred by chance. It involves the formulation of a null hypothesis, which is then tested against the alternative hypothesis. If the null hypothesis is rejected, it means that the observed effect is statistically significant and not likely to have occurred by chance. (Samuels et al., 2016)

3.10.2. Boxplot

A boxplot, also known as a box-and-whisker plot, is a graphical representation of a dataset that shows the distribution of the data along with outliers. The plot consists of a box with whiskers extending from both ends. The box in the middle represents the middle 50% of the data, with the left part of the box representing the 25th percentile and the right representing the 75th percentile. The line inside the box describes the median.

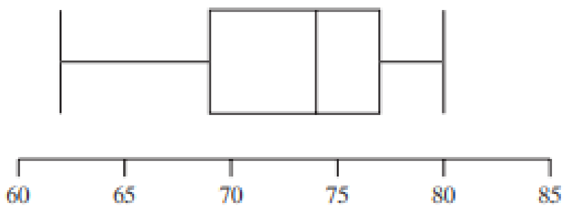


Figure 3. An example of a boxplot.
Image source: Samuels et al., 2016, p. 47.

The whiskers extending from either end of the box represent the range of the data. In some cases, the whiskers may represent a specific range, such as 1.5 times the interquartile range (IQR) above or below the box. Outliers, or data points that fall significantly outside the range of the box and whiskers, are represented by individual points beyond the whiskers.

A boxplot provides a rapid visual overview of the distribution of data. The median, represented by the line inside the box, allows us to determine the central point of the data.

3.10.3. Scatter plot

A scatter plot is a graphical representation of a dataset that displays the relationship between 2 variables. It is used to examine the association between the 2 variables, which can be positive (i.e. as one variable increases, the other variable also increases), negative (i.e. as one variable increases, the other variable decreases), or non-existent (i.e. no relationship between the variables).

In a scatter plot, each data point represents a pair of values for the 2 variables being analyzed. The data points are plotted on the graph with the x-coordinate and y-coordinate representing the respective values for the 2 variables.

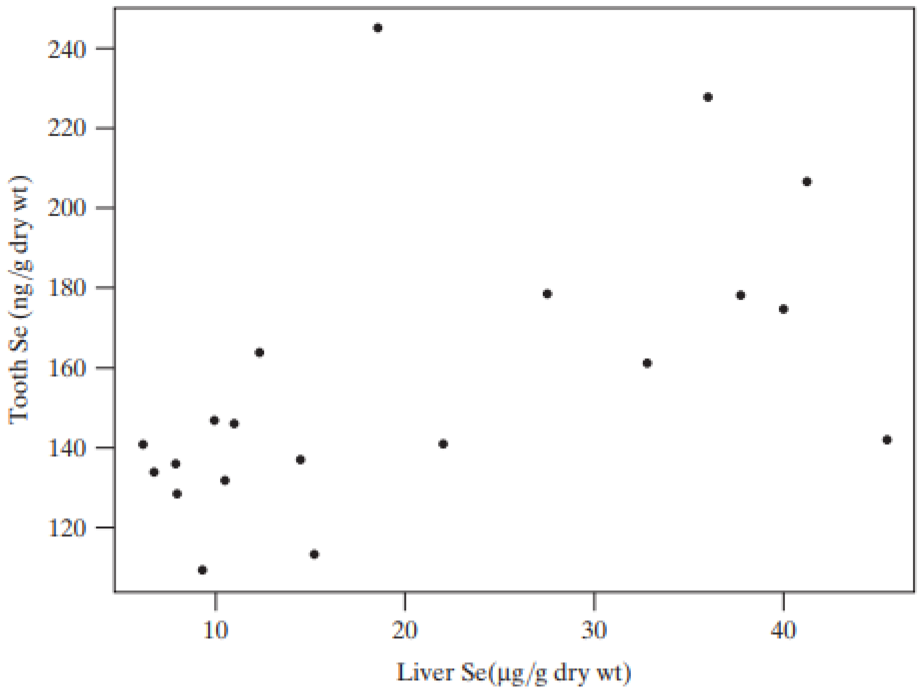


Figure 4. A scatterplot for tooth selenium concentration against liver selenium concentration of 20 belugas.
Image source: Samuels et al., 2016, p. 57.

Scatter plots can be enhanced with additional features such as a line of best fit, which is a straight line that represents the overall trend in the data. This line is typically calculated using regression analysis, which can help quantify the strength and direction of the correlation between the variables.

3.10.4. Histogram plot

A histogram plot is a graphical representation of the distribution of a dataset. It provides a visual depiction of the frequency or count of observations falling within different intervals, known as bins or buckets.

The x-axis of a histogram represents the range of values in the dataset, divided into equal-width intervals or bins. The y-axis represents the frequency or count of observations within each bin. The height of each bar in the histogram indicates the number of data points falling within that particular bin. Histograms are particularly useful for understanding the shape, central tendency, and spread of a dataset. They can reveal patterns such as skewness, modality (the number of peaks), and gaps or outliers.

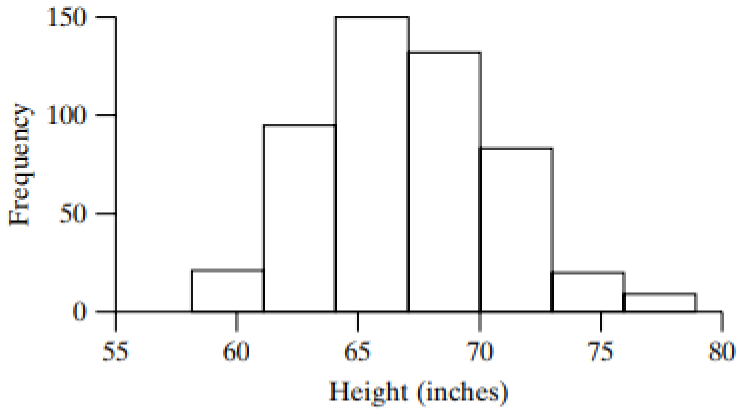


Figure 5. A histogram plot for heights of 510 college students.
Image source: Samuels et al., 2016, p. 44.

3.10.5. KDE plot

A KDE (Kernel Density Estimation) plot is a type of data visualization that estimates the probability density function of a continuous random variable. It provides a smoothed, non-parametric representation of the underlying distribution of the data. (Adriano Zanin Zambom & Dias, 2013)

Unlike a histogram that represents the data using discrete bins, a KDE plot uses a continuous curve to represent the density. The curve is formed by summing up a set of kernel functions, typically Gaussian kernels, placed at each data point. The bandwidth or smoothing parameter of the KDE determines the width of these kernels and affects the smoothness of the resulting curve.

KDE plots are useful for understanding the shape and characteristics of a dataset's distribution. They can reveal information such as the presence of multiple modes (peaks), skewness, and the overall smoothness of the distribution.

KDE plots are particularly effective when dealing with relatively large datasets or datasets that do not conform to a specific parametric distribution. They can also be used to compare distributions between different groups or subpopulations by overlaying multiple KDE curves on the same plot.

It's worth noting that KDE plots are based on estimation and can be influenced by the choice of bandwidth and kernel function. It is important to interpret KDE plots alongside other statistical measures and visualizations to gain a comprehensive understanding of the data.

In summary, KDE plots provide a smooth, continuous representation of the underlying distribution of data. They offer valuable insights into the shape and characteristics of a dataset, making them a useful tool for exploratory data analysis and visualization.

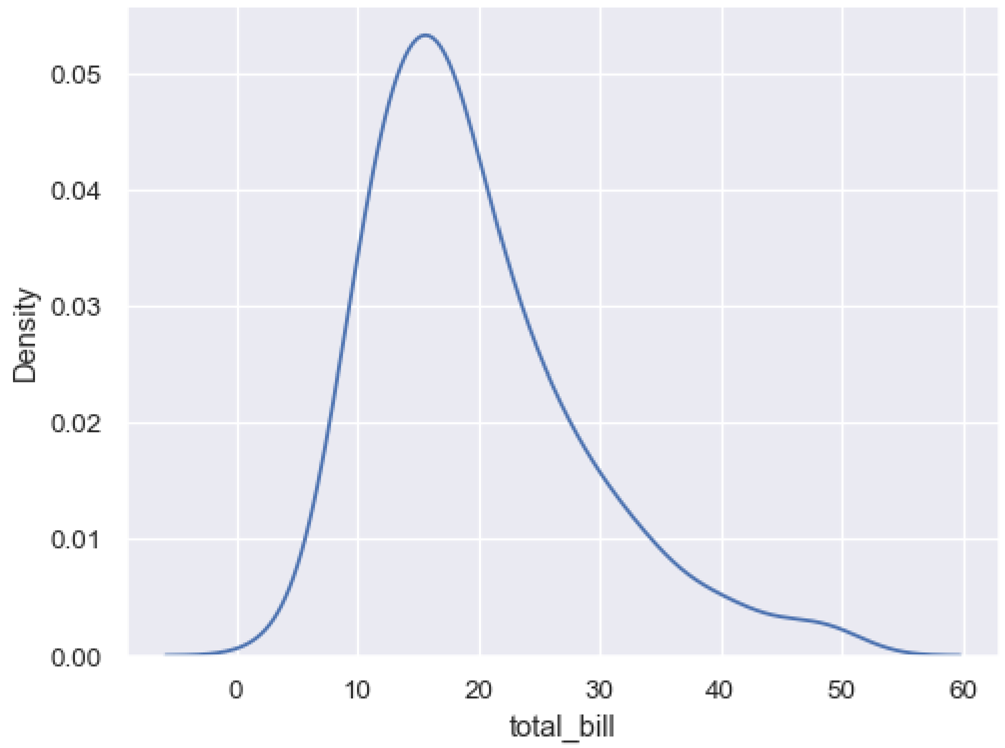


Figure 6. An example of a KDE plot from Seaborn's official documentation.
Image source: [seanborn.pydata.org]

3.10.6. Bar plot

The bar plot is a fundamental visual tool in statistical analysis that is used to represent categorical data. It provides a graphical representation of the frequency or distribution of data within different categories. The main purpose of a bar plot is to summarize and compare data across categories, allowing for easy interpretation and analysis. (Lane, 2016)

In a bar plot, each category is represented by a bar, and the height of the bar corresponds to the frequency, count, or proportion of data points within that category. The bars are usually arranged along the x-axis, while the y-axis represents the frequency or count. The length or width of each bar can vary, depending on the specific visualization requirements and the nature of the data being analyzed.

Bar plots are particularly useful when working with discrete or categorical variables, as they provide a clear visual representation of the distribution and relationships between different categories. The bar plot can also be customized with different colors, patterns, or labels to highlight specific features or comparisons within the dataset.

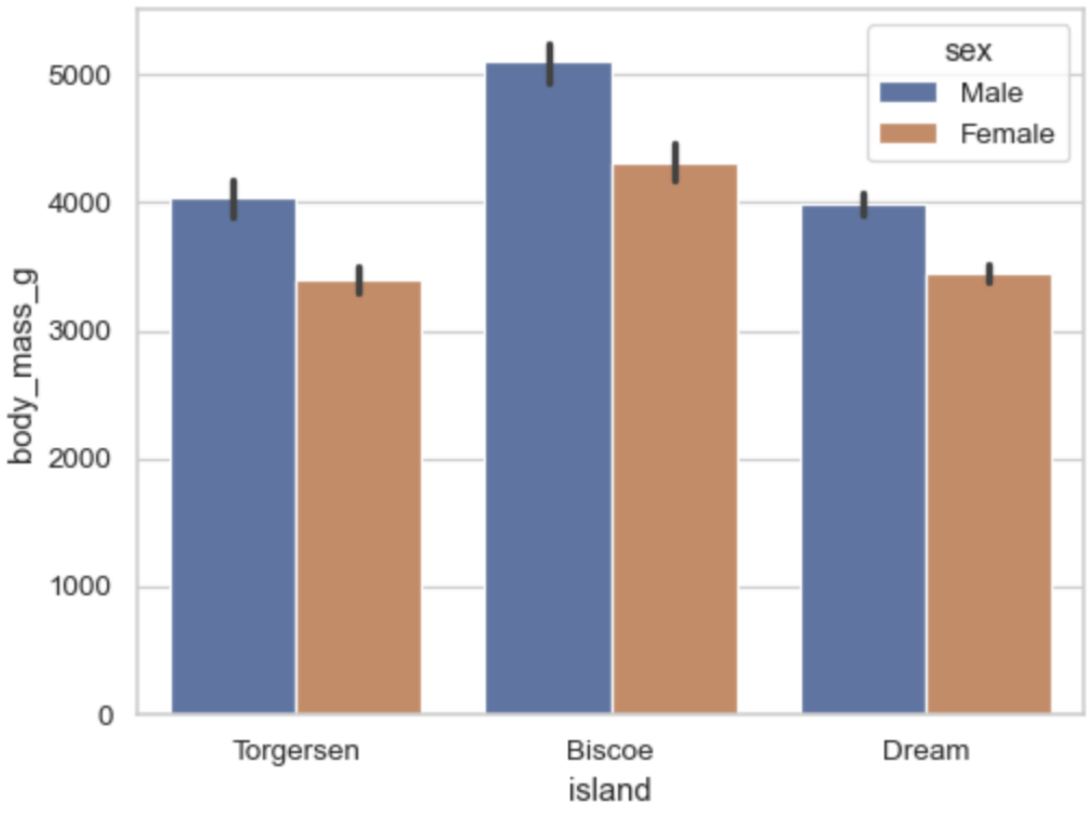


Figure 7. An example of a Bar plot from Seaborn's official documentation.
Image source: seanborn.pydata.org

3.10.7. Normal curve

The normal curve, also known as the Gaussian distribution or the bell curve, is a probability distribution that is widely used in statistics. It is a continuous distribution that is symmetrical around its mean, which is also its median. The shape of the normal curve is bell-shaped, with the highest point of the curve located at the mean, while the tails extend in both directions to infinity.

The normal curve is characterized by 2 parameters: the standard deviation and the mean. The mean determines the location of the center of the curve, while the standard deviation determines the shape of the curve. In a normal distribution, approximately 68% of the data falls within one standard deviation of the mean, 95% within 2 standard deviations, and 99.7% within 3 standard deviations.

The normal curve has several properties that make it useful in statistical analysis. For example, it provides a framework for estimating probabilities and making predictions about the behavior of large populations. It is also used in hypothesis testing, where it can be used to assess the likelihood of a particular outcome occurring by chance. Additionally, the normal curve is often used as a reference distribution for comparing other distributions in statistical analyses.

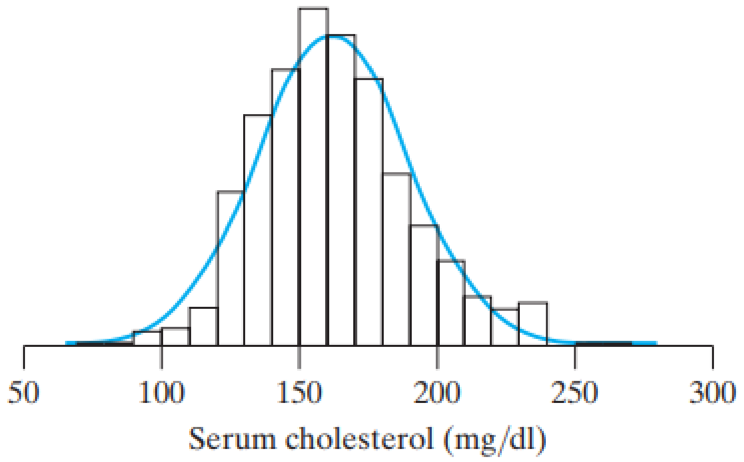


Figure 8. A normal distribution of serum cholesterol in 727 children (aged 12-14).
Image source: Samuels et al., 2016, p. 127.

3.10.8. Cochran's formula for sample size

Cochran's formula is a statistical formula used for determining the sample size. The formula is based on several factors, including the desired level of confidence, the margin of error, and the estimated proportion of the population with a certain characteristic or attribute. By inputting these variables into the formula, one can determine the sample size required to achieve their desired level of precision and margin of error.

One of the advantages of Cochran's formula is that it allows one to calculate the required sample size before conducting the study, which can help to save time and resources. Additionally, the formula can be adjusted based on the specific needs of the study, such as the level of confidence desired. Generally, the more precise the estimate, the larger the sample size needed. This is because smaller margins of error require larger sample sizes to achieve. (Uakarn et al., 2021)

The formula is $n = (p * (1 - p) * z\text{^}2) / e\text{^}2$, where each variable means the following:

- n: Sample size (the number of observations needed).
- p: Estimated proportion of the population with a specific characteristic or attribute.
- z: Z-score corresponding to the desired level of confidence (e.g., for a 95% confidence level, the z-score is approximately 1.96).
- e: Desired margin of error (the maximum allowable difference between the estimated proportion and the true population proportion).

3.10.9. A/B test

An A/B test is a statistical method used to compare 2 versions of a variable to determine if there is a statistically significant difference between them.

Generally, the A/B test typically involves dividing a sample into 2 groups: the control group (A) and the experimental group (B). The control group is exposed to the existing version of the variable, while the experimental group is exposed to a modified version. The purpose is to measure the performance or outcome of interest and determine if the changes made in the experimental group have a significant impact compared to the control group. (Adhikari et al., 2019)

In our case, we have used A/B testing to assess statistical significance and disregard statistical hypotheses. The null hypothesis (H0) assumes that there is no significant difference between the 2 groups, while the alternative hypothesis (H1) suggests that there is a significant difference. By collecting data from both groups and applying appropriate statistical tests, the test determines if the observed difference is statistically significant or likely due to random chance.

Statistical significance refers to the likelihood that an observed difference between variables is not due to random chance but is instead a meaningful and reliable finding. It allows us to determine whether the results obtained from a sample can be generalized to the larger population. Statistical significance is typically assessed using hypothesis testing, where we compare observed data to what we would expect to see by chance alone. The concept of statistical significance is important in A/B testing as it helps determine the reliability of the results. If the p-value (a measure of statistical significance) is below a predetermined threshold, the null hypothesis is rejected, indicating that there is a significant difference between the groups.

Conversely, if the p-value is above the threshold, the null hypothesis cannot be rejected, and no significant difference is observed.

On the other hand, statistical difference refers to the quantifiable distinction between 2 or more groups or variables. It represents the degree of divergence or variation between them. When comparing groups or variables, we often want to assess if there is a significant difference between them in terms of mean, proportion, effect size, or other measures of interest. In addition to statistical significance, it is also crucial to consider the practical or meaningful difference between the two groups. Even if a difference is statistically significant, it may not be practically significant or have real-world implications. Factors such as sample size, effect size, and practical significance should be taken into account when interpreting the results of an A/B test.

3.10.10. Normal distribution tests

Normal distribution tests, also known as tests of normality, are statistical tests used to assess whether a given data sample follows a normal distribution. The normal distribution also called the Gaussian distribution or bell curve, is a fundamental probability distribution widely used in statistical analysis.

There are several statistical tests available to evaluate the normality of data. Some common tests include the Shapiro-Wilk test, the Anderson-Darling test, and D'Agostino's K^2. These tests assess the null hypothesis that the data is normally distributed (Rani Das, 2016).

In these tests, the test statistic is computed based on the differences between the observed data and the expected values under a normal distribution. The resulting test statistic is then compared to a critical value or p-value to determine whether the data can be considered normally distributed. If the p-value is above a predefined significance level, we fail to reject the null hypothesis and conclude that the data is reasonably assumed to be normally distributed. If the p-value is below the significance level, we reject the null hypothesis and conclude that the data significantly deviates from a normal distribution. All 3 tests can be used to determine whether a data set can be reasonably assumed to be normally distributed.

The Shapiro-Wilk test is a popular normality test that assesses the deviation of a data set from a normal distribution. It calculates a W statistic that measures the agreement between the observed data and the expected values of a normal distribution. The test is sensitive to deviations in the tails of the distribution and is appropriate for small to moderate sample sizes.

D'Agostino's K^2 test is another normality test that assesses whether a data set follows a normal distribution. It calculates a K^2 statistic that measures the agreement between the observed data and the expected values of a normal distribution and takes into account the skewness and kurtosis of the data. The test is less sensitive to deviations in the tails of the distribution and is appropriate for larger sample sizes.

The Anderson-Darling test is a third normality test that assesses the goodness of fit of a data set to a normal distribution. It calculates an A^2 statistic that measures the agreement between the observed data and the expected values of a normal distribution and is sensitive to deviations in the tails of the distribution. The test is appropriate for small to moderate sample sizes.

4. Experiment Design

In this chapter, we describe each component that is part of our experiment. We also argue the decisions we have made along the way, and, generally, the process that led to our ultimate experiment design.

4.1. Node.js frameworks

When looking at the multitude of Node.js frameworks, we asked ourselves 2 questions: Which frameworks should be measured and how many should we include in the experiment?

To answer the first question, we considered the following criteria:

1. **Popularity**: We looked at GitHub, the biggest and most popular collection of open-source software. There, we looked for a list of the most popular frameworks ranked by GitHub stars[12], as this unity of measurement is used as a proxy for the general "like" button from the social media applications and represents the user's manifestation of interest related to a repository, project, or in this case, a Node.js framework (Borges & Valente, 2018).

2. **Compatibility with the server machine**: Our servers were hosted on a Raspberry Pi 4 Model B running Raspberry Pi OS. Further details on the Raspberry Pi are presented in the subchapter "4.2. Hardware".

3. **Compatibility with the database**: Our public relational database management system (RDBMS) was MSSQL. Further details on the database we have used are presented in the subchapter "4.3. Software".

For the second question, we considered the timeframe we had at our disposal for researching, developing, testing, measuring, comparing, analyzing, and writing; we thought that 2 frameworks would be too few to look into, while 3 would cover just enough to have a fair comparison, in our view. Hence, we decided to go with a total of 4, as this number also worked well with our initial plan to have a software skeleton of the project ready by February, while finalizing the final version of all APIs in March and April. We assessed that developing the code for a fifth framework would move our plan too late compared to our initial desired date for finishing the code and the report.

Considering all these factors, we selected the following frameworks for our measurements: Express.js, NestJS, Koa.js, and Fastify. All 4 are described below.

[12] Node.js API frameworks rank popularity on GitHub: https://github.com/vanodevium/node-framework-stars

4.1.1. Express.js

Express.js[13], often referred to simply as Express, is a popular open-source web application framework for Node.js. It provides a simple yet powerful set of features for building web applications and APIs, making it a popular choice for developers of all levels of experience. Express.js provides a range of built-in middleware packages that can perform common tasks such as serving static files, handling HTTP requests and responses, and managing user sessions. It also supports the use of third-party middleware software and plugins to extend its functionality. One of the key benefits of using Express.js is its flexibility, as it provides a minimal and unopinionated core that allows developers to build applications and APIs in various ways, depending on their needs and preferences. It is also highly modular, with many features provided as separate modules that can be installed via the CLI.

4.1.2. NestJS

NestJS[14] is a popular open-source backend framework for building scalable, maintainable, and enterprise-grade Node.js applications. It is built using TypeScript and combines elements of object-oriented programming (OOP), functional programming (FP), and functional reactive programming (FRP) to provide developers with a robust and flexible toolset for creating complex applications. NestJS is designed to be modular and flexible, allowing developers to create reusable and configurable modules that can be easily integrated into different applications. It also supports a range of libraries and tools for building RESTful APIs, real-time communication, GraphQL APIs, and more. One of the key features of NestJS is its use of dependency injection (DI) to manage the application's components and services. This allows for easy testing and reusability of code, as well as better code organization and maintainability. However, all the benefits come with a cost in terms of structure, as Nest will enforce the creation of certain files (such as services, modules, entities, and controllers) as seen in the MVC architectural pattern.

4.1.3. Fastify

Fastify[15] is an open-source web framework for building fast and efficient web applications with Node.js. It was created by Matteo Collina in 2016, and its primary goal is to provide a low-overhead framework for developers to build scalable web applications. Fastify is designed to be easy to use and feature-rich, while at the same time being lightweight and performant. One of the main features of Fastify is its high performance. Fastify is built from the ground up with a focus on speed and efficiency, using modern JavaScript features and advanced techniques to optimize performance. It also includes built-in support for HTTP/2, which enables faster and more efficient data transfer over the web. Fastify supports a wide range of plugins and middleware, which allows developers to easily extend and customize its functionality. It also includes support for async/await, which makes it easier to write clean and readable code that is easier to maintain. Furthermore, it contains a JSON schema validation, which can help to ensure that data input and output are valid and conform to a predefined schema.

[13] Express.js: https://expressjs.com/
[14] NestJS: https://nestjs.com/
[15] Fastify: https://www.fastify.io/

4.1.4. Koa.js

Koa.js[16] is a popular web framework for Node.js that is known for its minimalist and lightweight approach. It was created by the team behind the Express.js framework and was designed to improve on some of the limitations of Express. Koa.js is based on the use of middleware functions, which are functions that can be executed in a sequence to handle HTTP requests and responses. Koa.js allows for easy and flexible handling of asynchronous operations through the use of Promises and async/await syntax. Some of the key features of Koa include easy error handling through try/catch syntax, automatic error propagation, high performance through the use of generators, and optimized middleware handling. Koa.js is often used in building modern web applications, particularly those that require high-performance and scalable architectures, while benefiting from a growing community of developers and contributors, and is continually being updated and improved with new features and functionality.

4.2. Hardware

In this subchapter, we will describe the main hardware devices that are inherently more complex. Smaller devices, like SD cards or devices such as our laptops used for software development only, will not be elaborated upon, but just mentioned in the list of equipment.

4.2.1. Equipment

1. 2 x Raspberry Pi 4 Model B (client and server)
2. Siglent SPD3303X-3 programmable power supply
3. Sandisk 16GB MicroSD card
4. Sandisk 64GB MicroSD card
5. Lenovo M93z All-in-One (ThinkCentre) - Type 10AC
6. Macbook Pro M1 (for software development only)
7. MSI GS65 Stealth Thin 8-RF (for software development only)
8. Network Switch

4.2.2. Raspberry Pi

Raspberry Pi 4 Model B is a small single-board computer that was released in June 2019 by the Raspberry Pi Foundation. At the time of writing this thesis, it is the latest and most powerful model in the Raspberry Pi family and is designed to be used for a wide range of applications, including education, hobby projects, or even commercial projects.

The Raspberry Pi 4 Model B features a 1.5GHz quad-core ARM Cortex-A72 CPU, up to 8GB of RAM, Gigabit Ethernet, dual-band 802.11ac wireless, Bluetooth 5.0, 2 USB 3.0 ports, 2 USB 2.0 ports, 2 micro-HDMI ports (up to 4Kp60 supported), a 3.5mm audio jack, and a microSD card slot for storage. It also has a 40-pin GPIO header for connecting additional hardware and peripherals. The board is powered by a 5V USB-C power supply and can run a variety of operating systems,

[16] Koa.js: https://koajs.com/

including Raspberry Pi OS (formerly Raspbian), Ubuntu, and others. However, it does not come with any peripherals. (Eben Upton & Gareth Halfacree, 2016)

In our experiment, we used 2 Raspberry Pis, one for the server and one for the client. Both of them ran Raspberry Pi OS, an operating system flashed on 2 separate SD cards by using the Raspberry Pi Imager[17] software, and were powered via USB-C by a programmable power supply, described in more detail in the next subchapter.

As mentioned previously, we wanted to remove all possible internal and external factors that might influence energy consumption. Because of this, we ran both the client and the server in headless mode, without using any external monitor that would require additional computation (i.e. energy) for graphics. In addition, even if the Raspberry Pis had built-in Wi-Fi, we used Ethernet cables instead to connect both computers to a local network, in order to eliminate any chance of continuous access-point scanning, traffic on a specific channel, or in the worst-case scenario, random signal loss from the network.

The reasons for choosing Raspberry Pis as our main device to host and run the code are multiple:

1. Cost-effectiveness, making it an inexpensive solution for scientific experiments that require multiple devices for computing. The basic model costs only $35.

2. Small in size and easy to transport, which is ideal for field research where mobility is required.

3. Low power consumption, which is paramount for experiments that require long-term monitoring and data collection, as the device can be left running for extended periods without draining its power source.

4. Open-source software, allowing for an easy and quick setup.

Raspberry Pis can help make scientific experiments more accessible to a wider range of researchers and institutions. Since they are affordable and versatile, they can be used by small or underfunded research labs, as well as by individual researchers who may not have access to traditional scientific computing devices. Additionally, the abundance of online resources and tutorials makes it easier for researchers to get started with using them for scientific experiments.

[17] Raspberry Pi OS: https://www.raspberrypi.com/software/

4.2.3. Programmable Power Supply

The Siglent SPD3303X-E[18] is a programmable linear DC power supply that is designed for use in research, development, and production environments. The SPD3303X-E is a 3-channel power supply with a maximum output voltage of 32V and a maximum output current of 3.2A per channel. It features a high-resolution 4.3-inch LCD display that shows real-time voltage, current, and power information, as well as various other parameters such as setpoint values, overvoltage, and overcurrent protection settings.

The power supply is designed with a range of advanced features, including a low ripple and noise output, output timer and trigger functions, and voltage and current presets that can be saved for easy recall. It also has a built-in USB host and device interface for easy communication with a computer.

The main reason for choosing this power supply for our measurements was its availability and accessibility at the FabLab[19], a digital fabrication laboratory used for learning and research. What is more, the power supply can be used together with a piece of Java software, which we describe in chapter "4.3.3. TestController".

4.3. Software

In this chapter we describe the main software that we used for our thesis, aside from JavaScript and Node.js, which were mentioned in the chapter "3. Background". A list of all versions of our software package is attached in the chapter "10.2. Software versions".

4.3.1. RESTful APIs

We chose to use RESTful APIs for this experiment because they represent a standard that can be replicated and provide a structure by following a well-established predefined set of guidelines.

We started the implementation of the APIs by using pair programming, a software development technique originating from the Agile framework called Extreme Programming (Beck, 1999) which consists of 2 developers working together on the same computer. We decided to start with this approach as both of us could help one another by brainstorming various ideas, such as how to develop each API, how to establish the database connection for each framework or to discuss the details of the request validations. After the first API was finalized and we had a software template, we could then split the work for faster progress. Each API was developed by first initializing a new project as a Node.js project using the command "npm init -y" and then following the official documentation of each framework.

[18] Siglent SPD3303X-E: https://www.siglent.eu/product/1141026/siglent-spd3303x-e-linear-dc-3ch-programmable-power-supply
[19] FabLab: https://fablab.ruc.dk/

There are 4 fundamental operations for persistent storage, also known as CRUD. CRUD is an acronym that stands for create, read, update, delete, and all operations needed to be mapped to the HTTP methods of GET, POST, PUT, and DELETE. For GET, we implemented both a read-all method (i.e. where we read all entities) and also a read-by-id method, where we read only one entity by using its id.

We chose to create APIs not for one but for 2 different entities, one with a bigger size that has more fields (in our case, an employee from the employees table) and one with a smaller payload or fewer fields (a department from the departments table). The employees table has 6 columns, namely “emp_no”, “birth_date”, “first_name”, “last_name”, “gender” and “hire_date”, while departments consist only of a “dept_no” and a “dept_name”.

The reason for testing different entity sizes is to allow for a more comprehensive analysis of the energy consumption behavior of the system and the device being tested. By measuring the energy consumption for 2 different payload sizes, we can obtain a better understanding of how the energy consumption is affected by the size of the payload being transmitted or processed. In many real-world scenarios, the size of the data can vary significantly, and as this happens, the system may need to allocate additional resources to process it effectively, having an influence on energy consumption. In addition, using 2 different payload sizes can also help to validate the accuracy of the energy consumption measurements. By comparing the energy consumption values, we can ensure that the measurements are consistent and reliable, which is important when concluding the energy optimization strategies (in other words, what framework to choose for developing Application Programming Interfaces).

The general endpoints that we developed for each framework are described below. All endpoints use JSON as the format for sending and receiving data and each endpoint has a uniform interface, starting with the **/api/** path, followed by the name of the entity, which is a noun in plural form (**/employees** or **/departments**):

- **GET “/api/employees”** to retrieve the first 10 employees from the database. We decided to retrieve only the first 10 employees due to the large size of the table (300024 entries in total). It is also common practice to use pagination and filter the results for large sizes (Masse, 2011).

- **GET “/api/employees/:id”** to read an employee by ID. Returns a “404 Not Found” error code if the ID does not exist.

- **POST “/api/employees”** to create a new employee. The “emp_no” is auto-incremented by taking the total number of employees in the database and increasing it with 1. For the body that was being sent with the request, we needed to add some validation due to inherent database constraints. Therefore, we added the validation on the “birth_date” and “hire_date” to have the “yyyy-mm-dd” format. We also added validations on “first_name” which has a minimum of 2 characters and a maximum of 14, while the “last_name” has a minimum of 2 characters and a maximum of 16. What is more, the gender could only be one character, either “M” for males or “F” for females.

- **PUT “/api/employees/:id”** to update an employee by ID. Returns a “404 Not Found” error code if the ID does not exist and includes the same validations as for creating an employee.

- **DELETE “/api/employees/:id”** to remove an employee by ID. Returns a “404 Not Found” error code if the ID does not exist.

- **GET “/api/departments”** to retrieve all 9 departments from the database.

- **GET “/api/departments/:id”** to read a department by ID. Returns a “404 Not Found” error code if the ID does not exist.

- **POST “/api/departments”** to create a new department. The “dept_no” is auto-incremented, by taking the total number of departments in the database and increasing it with 1. For the body that was being sent with the request, we needed to add some validation due to inherent database constraints. Therefore, we added the validation on the “dept_name” for being a required and unique field.

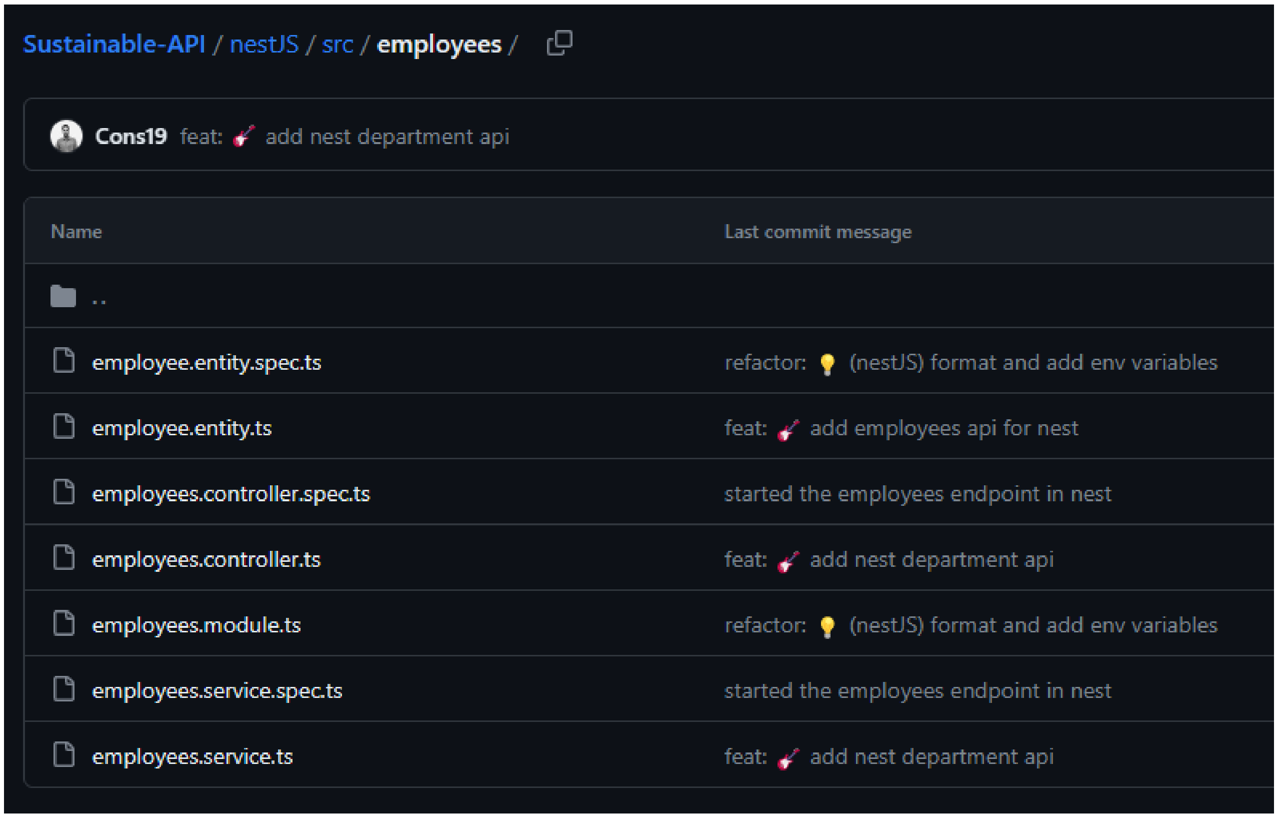
- **PUT “/api/departments/:id”** to update a department by ID. Returns a “404 Not Found” error code if the ID does not exist and includes the same validations as for creating a department.

- **DELETE “/api/departments/:id”** to remove a department by ID. Returns a “404 Not Found” error code if the ID does not exist.

The ports used for each framework are 3000 for Express, 3001 for Nest, 3002 for Fastify, and 3003 for Koa.

Regarding the folders and files, 3 frameworks (Express, Koa, and Fastify) follow a simple structure by having a main JavaScript file (called “index.js” in all cases) that is executed by Node. This allows developers to include any other third-party libraries or technologies (like middlewares), while also having full liberty over the project structure.

Nest, however, is the only opinionated framework that requires a specific way of structuring files and folders. The structure is similar to other MVC frameworks, in the sense that we have a clear separation between entities (e.g. employees or departments), services (classes that deal with the logic for each endpoint), and controllers (the classes that handle each request by sending it to the services and returning the response to the client). When creating these files, Nest automatically creates specification files with the “.spec” suffix that are designed for testing purposes. Each entity is considered a module, and each separate module is imported into the main “app.module.ts” file. In Figure 9, you can see the Nest files related to the employee entity.


Sustainable-API / nestJS / src / employees /

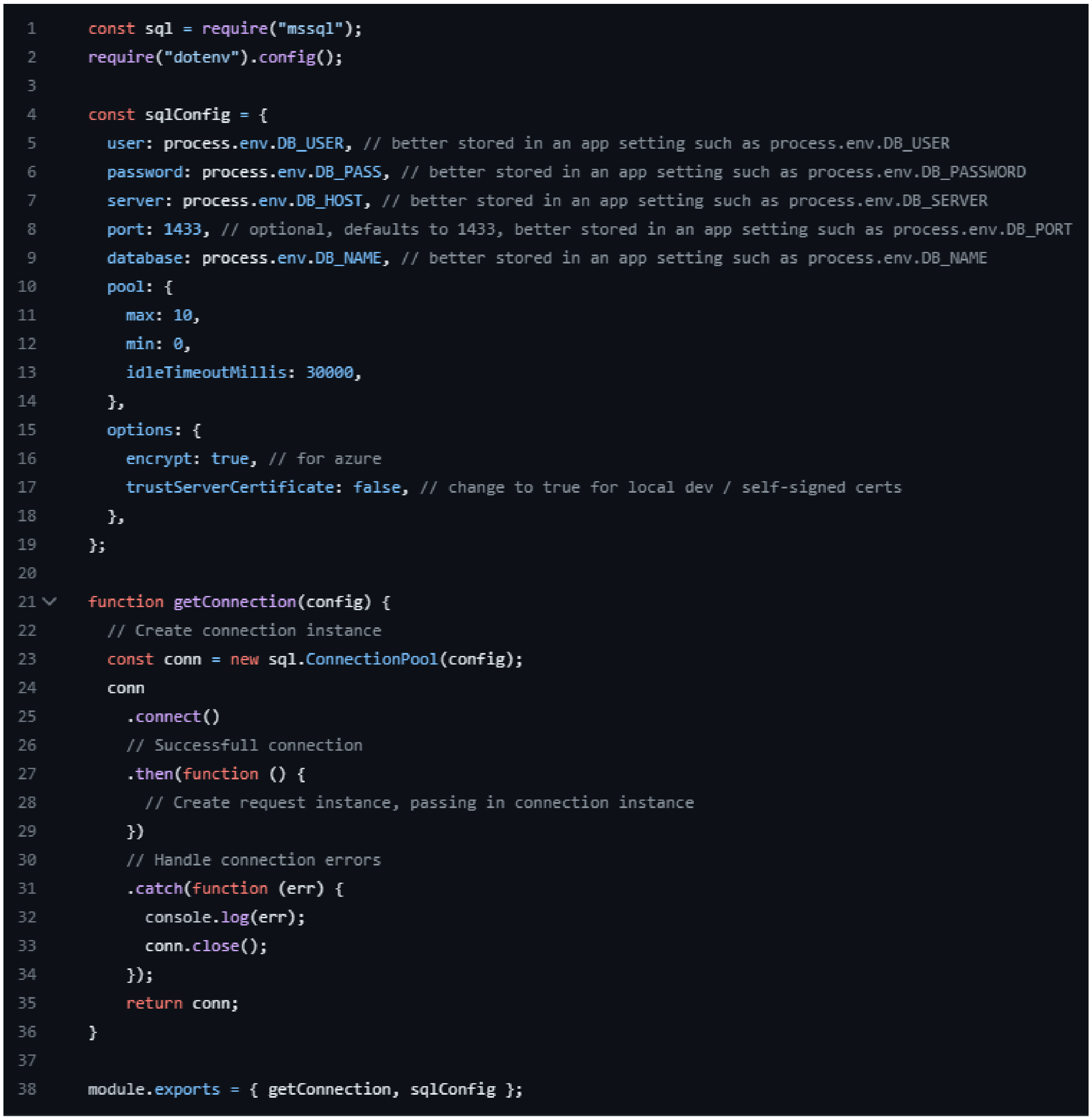
Cons19 feat: add nest department api

Name	Last commit message
..	
employee.entity.spec.ts	refactor: 💡 (nestJS) format and add env variables
employee.entity.ts	feat: add employees api for nest
employees.controller.spec.ts	started the employees endpoint in nest
employees.controller.ts	feat: add nest department api
employees.module.ts	refactor: 💡 (nestJS) format and add env variables
employees.service.spec.ts	started the employees endpoint in nest
employees.service.ts	feat: add nest department api

Figure 9. Employee-related files for Nest under the src/employees directory. The screenshot is taken from our GitHub repository.

To facilitate the connection to the public database, we took a systematic approach by establishing a dedicated folder named “db-connection”. Within this folder, a component called “db-connection.js” was created to encompass the primary logic required for establishing a connection to Microsoft Azure, where our database resides. This file served as a centralized hub for connecting to the database, eliminating the need for redundant code scattered across multiple frameworks. By adopting this consolidated approach, all frameworks were able to import the “db-connection.js” file, thus accessing the necessary connection logic. This approach not only streamlined the development process but also ensured that any modifications to the connection logic or changes of the database engine could be made in a single location, minimizing the risk of inconsistencies or errors that could arise from duplicating the same code across different frameworks.

For security purposes, the authentication credentials (i.e.the username, password, the host, and the database) are not present nor visible in our GitHub repository (see Figure 10 below) but are rather stored locally in environment variables, in a file with the “.env” extension. An environment variable is a dynamic value that can be set outside of the code and used within the application and typically stores configuration information, such as API keys or authentication credentials.



```
const sql = require("mssql");
require("dotenv").config();

const sqlConfig = {
  user: process.env.DB_USER, // better stored in an app setting such as process.env.DB_USER
  password: process.env.DB_PASS, // better stored in an app setting such as process.env.DB_PASSWORD
  server: process.env.DB_HOST, // better stored in an app setting such as process.env.DB_SERVER
  port: 1433, // optional, defaults to 1433, better stored in an app setting such as process.env.DB_PORT
  database: process.env.DB_NAME, // better stored in an app setting such as process.env.DB_NAME
  pool: {
    max: 10,
    min: 0,
    idleTimeoutMillis: 30000,
  },
  options: {
    encrypt: true, // for azure
    trustServerCertificate: false, // change to true for local dev / self-signed certs
  },
};

function getConnection(config) {
  // Create connection instance
  const conn = new sql.ConnectionPool(config);
  conn
    .connect()
    // Successfull connection
    .then(function () {
      // Create request instance, passing in connection instance
    })
    // Handle connection errors
    .catch(function (err) {
      console.log(err);
      conn.close();
    });
    return conn;
}

module.exports = { getConnection, sqlConfig };
```

Figure 10. The contents of the "db-connection.js" file.
The screenshot is taken from our GitHub repository.

One of the constraints of the RESTful architectural style is caching. Caching allows the responses from a web service to be stored in a cache, which can then be used to satisfy subsequent requests for the same resource without the need for the server to generate a new response. While caching can provide significant performance benefits by reducing the number of requests that need to be handled by the server, it can also impact energy consumption. Specifically, caching can reduce energy consumption in APIs by reducing the number of requests that need to be processed by the server. This works by storing the response of a request on the client side or an intermediate server, such as a CDN or a reverse proxy. When the same request is made again, the cached response can be served without having to go through the entire process of processing the request on the server. This reduces the load on the server and can save energy.

In this context, it is paramount to isolate the impact of caching to accurately measure the energy consumption of the API itself. This may require avoiding the use of caching in the software implementation, even though caching is a fundamental part of the RESTful architectural style. By avoiding caching in the API implementation, the energy consumption measurements can be more accurately attributed to the framework itself, rather than being influenced by the caching mechanism. This is something that we have accounted for by not using any caching-related middleware in our frameworks, but simply using the bare minimum in terms of libraries and classes to cover our functionality.

However, it is important to note that the decision to avoid caching in the implementation of an API should be made with careful consideration concerning the potential impact on performance and scalability. Caching can provide significant performance benefits, and avoiding it may result in slower response times and increased server load. Therefore, the decision to avoid caching should be based on a careful assessment of the trade-offs between energy consumption, performance, and scalability.

All the code is available on GitHub via this link: https://github.com/Cons19/Sustainable-API.

4.3.2. Database

The database we have decided to use for our API calls is “Employees Sample Database” which can be found here: https://dev.mysql.com/doc/employee/en/. It is a publicly available database in the official documentation from MySQL[20] created by Patrick Crews and Giuseppe Maxia and it is designed to provide a realistic set of data for software development and testing purposes within database management systems, particularly the relational database management system MySQL.

The database consists of 6 tables that contain data related to employees and their employment in a fictitious company. These tables are employees, departments, salaries, titles, department managers, and the departments for each employee. The size is 160MB and has 4 million records in total. The schema with all the fields can be seen below, in Figure 11. In addition to the base data, the database also includes a suite of tests that can be executed across the test data to ensure the integrity of the data that was loaded. This should help ensure the quality of the data during the initial load or to ensure that no changes have been made to the database during testing.

[20] GitHub link for the SQL files: https://github.com/datacharmer/test_db

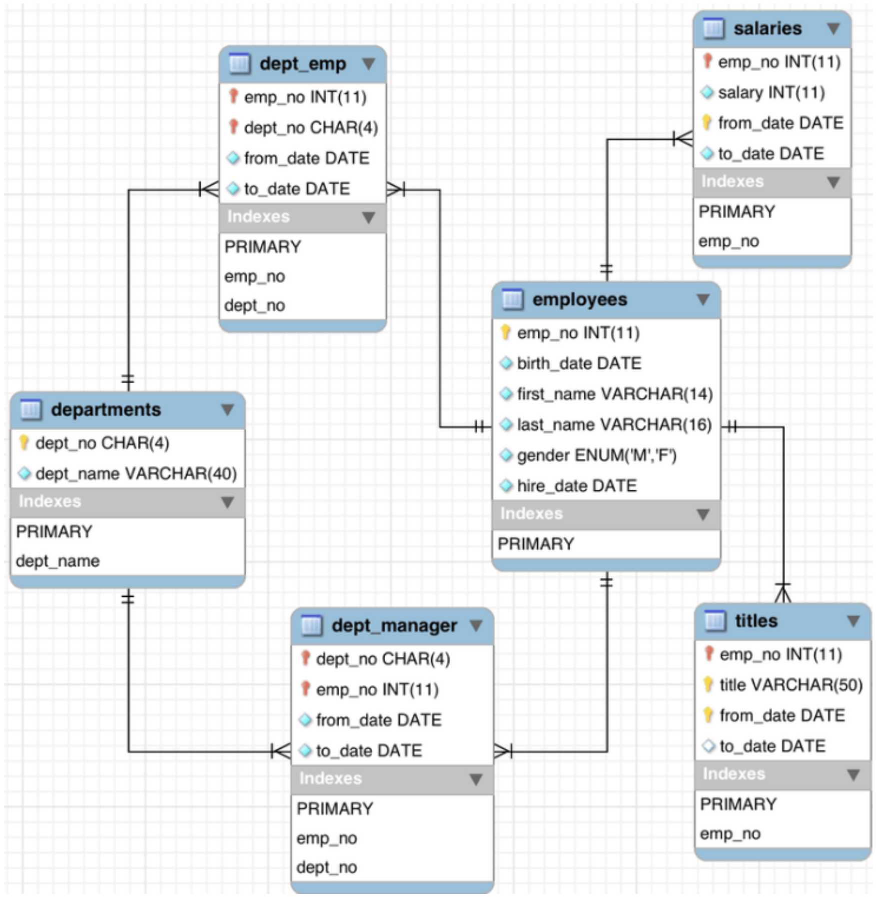


Figure 11. "Employees" database schema.
Image source: https://dev.mysql.com/doc/employee/en/sakila-structure.html.

There are several reasons for choosing a public and relational database (Harrington, 2016) for our research:

1. Access to a large amount of data: Public databases can contain vast amounts of data that are freely available for researchers to use. This can save a considerable amount of time and effort in collecting and curating your data.

2. Reproducibility and transparency: Using a public database can increase the transparency and reproducibility of our thesis findings. Other researchers can easily access and verify the data in their studies, which can help to increase the credibility of our experiment.

3. Cost-effectiveness: Using a public database can be a cost-effective way to obtain and use data without having to allocate resources for data collection and curation.

4. Structured data: Relational databases are designed to handle structured data, which is data that fits neatly into tables with clearly defined relationships between them. This is particularly useful for applications that require complex queries and need to maintain data consistency and accuracy.

5. Data integrity: Relational databases provide mechanisms for ensuring data integrity, such as foreign keys and constraints, which help prevent data from being added or deleted in an inconsistent or invalid way, especially in a public API where many clients are allowed to submit requests and modify the data.

6. Query language: Relational databases use SQL, a standardized query language that allows for powerful and flexible querying of data. This makes it easy to retrieve specific data subsets from large and complex datasets.

As previously stated, we wanted our development setup to reassemble a real scenario. One of the implications for this is for the database to be accessed via the network (i.e. deployed on the cloud) and not hosted locally on the server machine. Since Roskilde University put to our disposal Microsoft licenses that provide us with credits worth $100, we decided to deploy the "Employees Sample Database" to Microsoft Azure[21], one of the popular cloud computing platforms besides Amazon Web Services and Google Cloud.

We started by creating the SQL server with an Azure subscription, followed by creating the SQL database using the basic tier of 2GB of memory and 5 Database transaction units (DTUs[22]), this being the only tier that would fit into our $100 budget. Unfortunately, deploying the database to Microsoft Azure meant that we were forced to go with the MSSQL database. This influenced the overall experiment in the initial stages as one of the frameworks that we had initially picked, strapi[23], did not support MSSQL. Due to this, we had to choose the next most popular framework and discard strapi from our measurements due to software incompatibility.

4.3.3. TestController

TestController[24] is a Java application that we used to collect data for our energy measurements. All the necessary files needed to run TestController were already located on the computer connected to the power supply at the FabLab (i.e. device number 5 from the "4.2.1. Equipment" list).

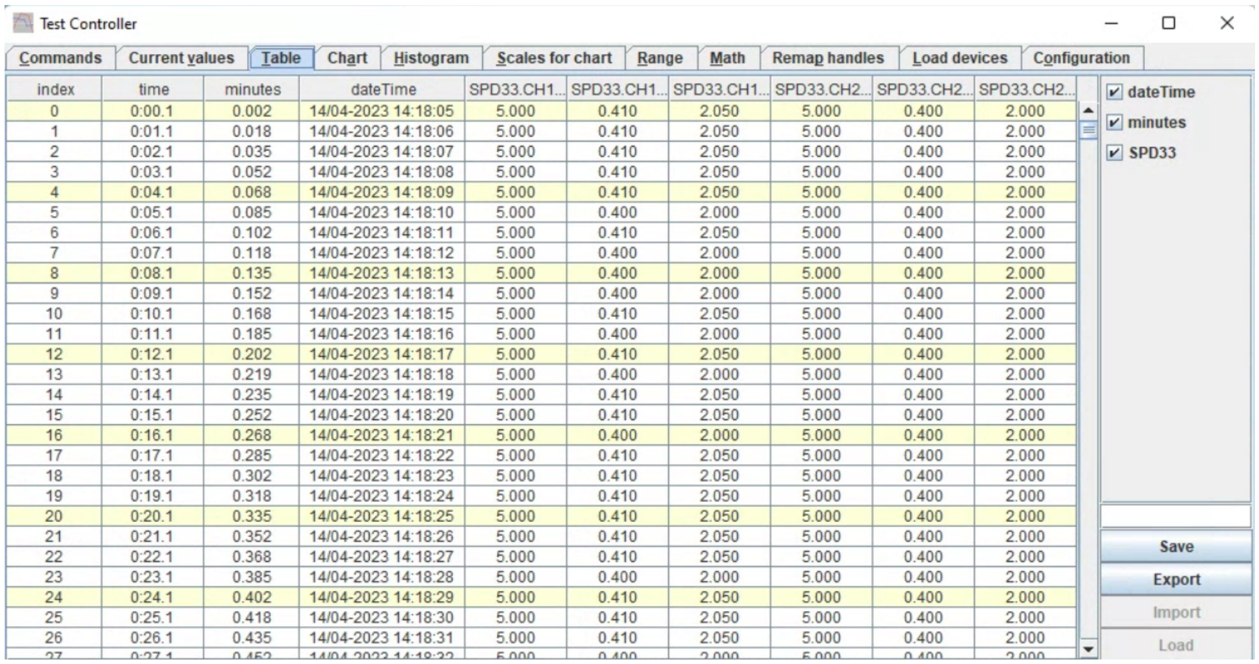
TestController allows for SCPI commands, a standardized set of commands used to operate and control scientific equipment. These commands are sent through a communication interface, such as an Ethernet, to manage parameters, perform measurements, and retrieve data from the instrument. However, in our case, we simply used the GUI of the software, as this proved to be enough for our setup.

[21] Microsoft Azure's homepage: https://azure.microsoft.com/en-us
[22] Database transaction units:
https://learn.microsoft.com/en-us/azure/azure-sql/database/service-tiers-dtu?view=azuresql
[23] Strapi's homepage: https://strapi.io/
[24] TestController page: https://lygte-info.dk/project/TestControllerIntro%20UK.html

When executed, the program displays a window with multiple tabs (or submenus). The menus that we have used are the following:

1. “Current values” shows the parameters (e.g. voltage, current, power) that were preconfigured in the power supply.

2. “Table” displays live data in a table format of the energy consumption taken from the devices connected to the power supply.



index	time	minutes	dateTime	SPD33.CH1...	SPD33.CH1...	SPD33.CH1...	SPD33.CH2...	SPD33.CH2...	SPD33.CH2...
0	0:00.1	0.002	14/04-2023 14:18:05	5.000	0.410	2.050	5.000	0.400	2.000
1	0:01.1	0.018	14/04-2023 14:18:06	5.000	0.410	2.050	5.000	0.400	2.000
2	0:02.1	0.035	14/04-2023 14:18:07	5.000	0.410	2.050	5.000	0.400	2.000
3	0:03.1	0.052	14/04-2023 14:18:08	5.000	0.410	2.050	5.000	0.400	2.000
4	0:04.1	0.068	14/04-2023 14:18:09	5.000	0.410	2.050	5.000	0.400	2.000
5	0:05.1	0.085	14/04-2023 14:18:10	5.000	0.400	2.000	5.000	0.400	2.000
6	0:06.1	0.102	14/04-2023 14:18:11	5.000	0.410	2.050	5.000	0.400	2.000
7	0:07.1	0.118	14/04-2023 14:18:12	5.000	0.400	2.000	5.000	0.400	2.000
8	0:08.1	0.135	14/04-2023 14:18:13	5.000	0.400	2.000	5.000	0.400	2.000
9	0:09.1	0.152	14/04-2023 14:18:14	5.000	0.400	2.000	5.000	0.400	2.000
10	0:10.1	0.168	14/04-2023 14:18:15	5.000	0.410	2.050	5.000	0.400	2.000
11	0:11.1	0.185	14/04-2023 14:18:16	5.000	0.400	2.000	5.000	0.400	2.000
12	0:12.1	0.202	14/04-2023 14:18:17	5.000	0.410	2.050	5.000	0.400	2.000
13	0:13.1	0.219	14/04-2023 14:18:18	5.000	0.400	2.000	5.000	0.400	2.000
14	0:14.1	0.235	14/04-2023 14:18:19	5.000	0.410	2.050	5.000	0.400	2.000
15	0:15.1	0.252	14/04-2023 14:18:20	5.000	0.410	2.050	5.000	0.400	2.000
16	0:16.1	0.268	14/04-2023 14:18:21	5.000	0.400	2.000	5.000	0.400	2.000
17	0:17.1	0.285	14/04-2023 14:18:22	5.000	0.410	2.050	5.000	0.400	2.000
18	0:18.1	0.302	14/04-2023 14:18:23	5.000	0.410	2.050	5.000	0.400	2.000
19	0:19.1	0.318	14/04-2023 14:18:24	5.000	0.410	2.050	5.000	0.400	2.000
20	0:20.1	0.335	14/04-2023 14:18:25	5.000	0.410	2.050	5.000	0.400	2.000
21	0:21.1	0.352	14/04-2023 14:18:26	5.000	0.410	2.050	5.000	0.400	2.000
22	0:22.1	0.368	14/04-2023 14:18:27	5.000	0.410	2.050	5.000	0.400	2.000
23	0:23.1	0.385	14/04-2023 14:18:28	5.000	0.400	2.000	5.000	0.400	2.000
24	0:24.1	0.402	14/04-2023 14:18:29	5.000	0.410	2.050	5.000	0.400	2.000
25	0:25.1	0.418	14/04-2023 14:18:30	5.000	0.410	2.050	5.000	0.400	2.000
26	0:26.1	0.435	14/04-2023 14:18:31	5.000	0.410	2.050	5.000	0.400	2.000

Figure 12. A screenshot from the “Table” menu of the TestController.jar after logging is started.

3. “Chart” contains the same data as from the table format, but here it is shown in a more visual way.

4. “Load devices” is used for managing the power supplies visible on the computer where TestController is run from.

5. “Configuration” is a menu used for various user-made configurations such as image size resolutions, font size for the text, or how to handle data overloading.

During data logging, TestController sends requests to the power supply at the configured frequency rate. Our logging frequency was set to one second. After finalizing the measurements, we exported all the results from the “Table” menu to a CSV format, which was later used for statistical analysis.

4.3.4. Automated script

To ensure the replicability of our measurements, we needed a way to reproduce them every time in a consistent way. Manually doing so would prove impossible for human beings, thus, we needed to leverage the power of scripting in order to mimic the sequence of our API method calls.

We had to call each endpoint method (i.e. GET all, GET by ID, POST, PUT, and DELETE) for each entity (i.e. employees and departments) as well as for each framework (i.e. Express, Nest, Fastify, and Koa). We had to choose between developing our script or using an existing benchmarking tool for automating the execution of requests. One such popular tool for Node.js is autocannon[25]. After experimenting with a few automation programs, we decided to create our Bash script because it would allow us to customize it and change it to better suit our needs.

In order to ensure the accuracy and validity of our energy consumption measurements, we followed the methodologies and techniques outlined in various articles on energy measurements (Cruz & Abreu, 2021) (Patrou et al., 2021) (Kirkeby et al., 2022). By adhering to these approaches, we aimed to obtain unbiased and reliable data that is widely accepted in the field of energy consumption research. Consequently, we made an effort to rule out any outside influences that might have affected the consumption of our Raspberry Pis by closing all applications, turning the Wi-Fi, Bluetooth, and the automatic updates off, and using only the ethernet connection to the network switch. As mentioned before, we ran the server and the client in headless mode by starting the server, the client and executing the script through SSH, without using any external monitors. We have also prepared separate files used for the API interaction, with all possible URLs for the endpoints, together with ".txt" files used for the body POST and PUT requests. At the same time, we have fully automated the executions of the tests, including the number of iterations between calls and the one-minute breaks between every 30 requests, for allowing the CPU to cool down and prevent overheating. We decided to start and end each experiment with a baseline measurement by waiting 5 minutes to make sure no background processes were active.

Activity	**Time (in s)**	**Iterations**
Baseline waits (before and after each experiment)	300	2 per experiment
API calls	Logged by TestController	30 (consecutive)
Breaks after each set of the 30 runs	60	30

Table 1. Overview of the automated experiment design.

[25] Link to GitHub: https://github.com/mcollina/autocannon

We created one Bash script where we used curl[26] to make all requests from the client to the server by reading each URL from the separate files. We had to split the URLs into 8 files due to the aforementioned 4 frameworks and 2 entities for each framework. Therefore, an experiment for 1 framework entailed 30 consecutive runs with a 1-minute break in between, each run having 30 iterations times the 5 endpoints following the same order as in Table 2, for each of the 2 entities. To be able to match the execution of the requests on the server with the calls that were sent from the client, we also kept a log on the client side with the iteration number, the date, and the time before and after each of the 30 runs (seen further below in Figure 14).

Experiment Number	**Framework**	**API calls**
1	Express.js	30 * GET, GET by id, POST, PUT, DELETE (employees & departments) back-to-back
2	NestJS	30 * GET, GET by id, POST, PUT, DELETE (employees & departments) back-to-back
3	Fastify	30 * GET, GET by id, POST, PUT, DELETE (employees & departments) back-to-back
4	Koa.js	30 * GET, GET by id, POST, PUT, DELETE (employees & departments) back-to-back

Table 2. Overview of all experiments.

When executing the script, we wanted to keep the database in the same state for each experiment. As a result, we have designed the script such that after each iteration, the database is reverted to the initial state. This is done by creating a new entity in POST, which we then modify in PUT, and remove in the DELETE request, keeping the database with the original data that was imported from GitHub. This occurs for both the employees and the departments during each experiment. This is to ensure that all frameworks work under the same conditions and are compared objectively in the same environment.

The full Bash script can be found in chapter “10.4. Bash script”, but essentially, the contents of the Bash script can be described in a pseudocode-like manner as follows:

1. Remove any previous log file for the previous runs, if any.

2. Wait for 300 seconds.

[26] Curl: https://curl.se/

3. Execute the run_files function for each file, i.e. express/employees.txt, express/departments.txt, nest/employees.txt, nest/departments.txt, fastify/employees.txt, fastify/departments.txt, koa/employees.txt, koa/departments.txt.

4. The run_files function:
    a. Iterate with i from 1 to 30:
        i. Log timestamp before the calls.
        ii. Iterate with j from 1 to 30:
            While we can read requests from the .txt files:
                a. Echo the command (URL).
                b. Send the curl request.
        iii. Echo the value of each i.
        iv. Log timestamp after the calls.
        v. Wait for 60 seconds.
    b. Wait for 300 seconds.

5. Echo “done”.

4.4. Physical setup

We had both Raspberry Pis powered by the programmable power supply. Another computer that was communicating with the same power supply via the network was the Lenovo All-in-One desktop computer from the FabLab, from where we started the measurements and extracted the data by using TestController. All devices were connected to the local network via ethernet cables so that they could communicate with each other. An overview of the hardware setup can be seen below:

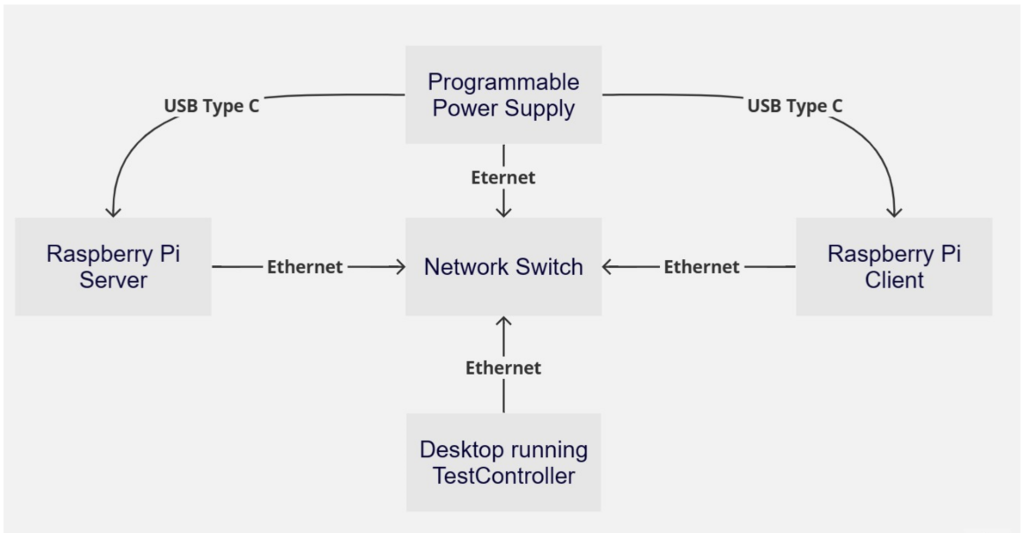


Figure 13. A diagram of our physical setup.
Source: Miro[27].

[27] A free visual collaboration tool: https://miro.com/

5. Data Preparation and Analysis

In this section, we describe the process we have used to prepare the data for the analysis. All the Python code used for the statistical analysis can be found on our GitHub repository, linked at the end of the "4.3.1 RESTful APIs" subchapter.

5.1. Preparation

After our measurements were completed, we had 2 main data sources.

The first one was a CSV file with the logs for each measurement generated by TestController. This file was saved on the desktop computer from FabLab. This file had fields (as seen in "4.3.3. TestController") such as the timestamps in Unix epoch format of the measurements, the voltage in Volts, the current in Amperes, and the power in Watts measured for both the client and the server, as both Raspberry Pis were connected to the same power supply via different channels.

The second data source was the log file created by our script, which was stored on the client Raspberry Pi from where we initiated the requests. The Bash script logged the timestamps in British Summer Time format before and after each set of requests was made. As mentioned before, an individual set, sample or measurement included 30 consecutive calls multiplied by the number of the HTTP requests, which in total was 5, resulting in batches of 150 consecutive requests. All of this was executed 31 times spaced with 1-minute breaks in between.

To align the data, we had to look at both log files and compare the timestamps. A manual approach was used to split the final CSV file exported from TestController into smaller, individual ones pertaining to each run (in other words, having only the necessary data for the time window when an interaction between the client and the server occurred). During our data analysis, we discovered that an automated solution for splitting a larger CSV file into smaller ones was excluding some data points that were part of the actual runs, due to a couple of measurement values being equal to the idle power consumption, thus not allowing for a clear separation between the relevant data points and those that were part of the breaks or idle consumption. Hence, our automated solution was not counting the data correctly, which could have led to incorrect conclusions. Despite the potential for human error, we argue that a manual approach was the only viable solution in this case.

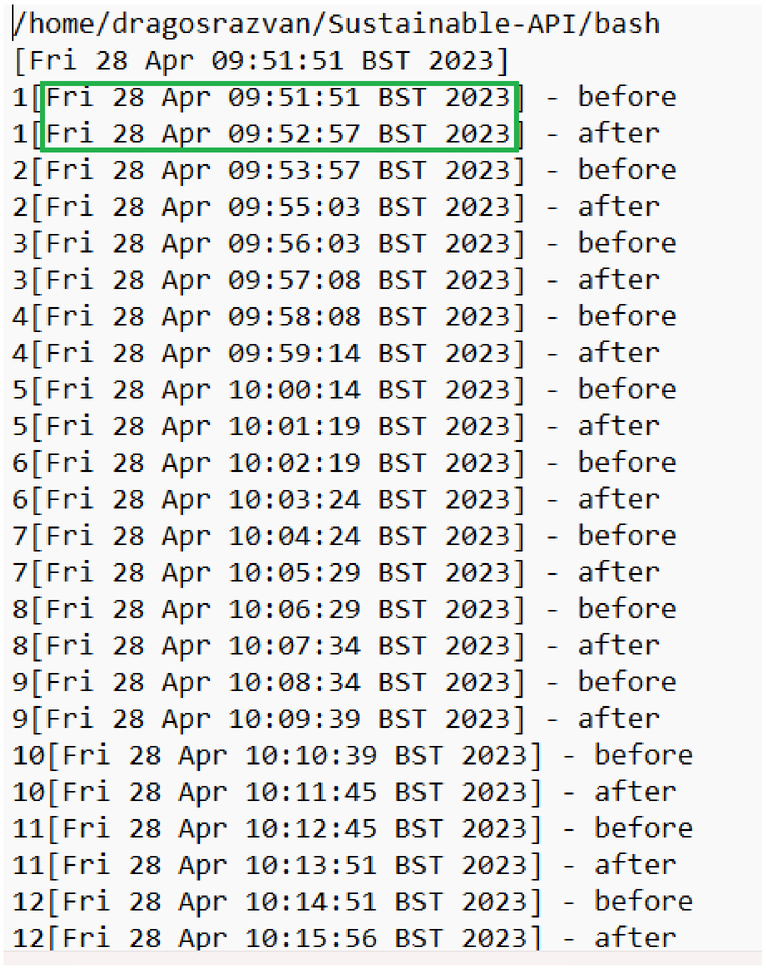
The process we used for preparing the data was the following:

1. While being on the client Raspberry Pi:
    a. Go to the main folder of each framework.
    b. Navigate to the "results" subfolder.
    c. Open the log file.

2. Identify the time before and after each run:

a. Ignoring the first 2 lines, as those are used for testing, ensuring the current working directory and that we can print the date

b. Each run is identified by looking at the index that is printed before the dates.



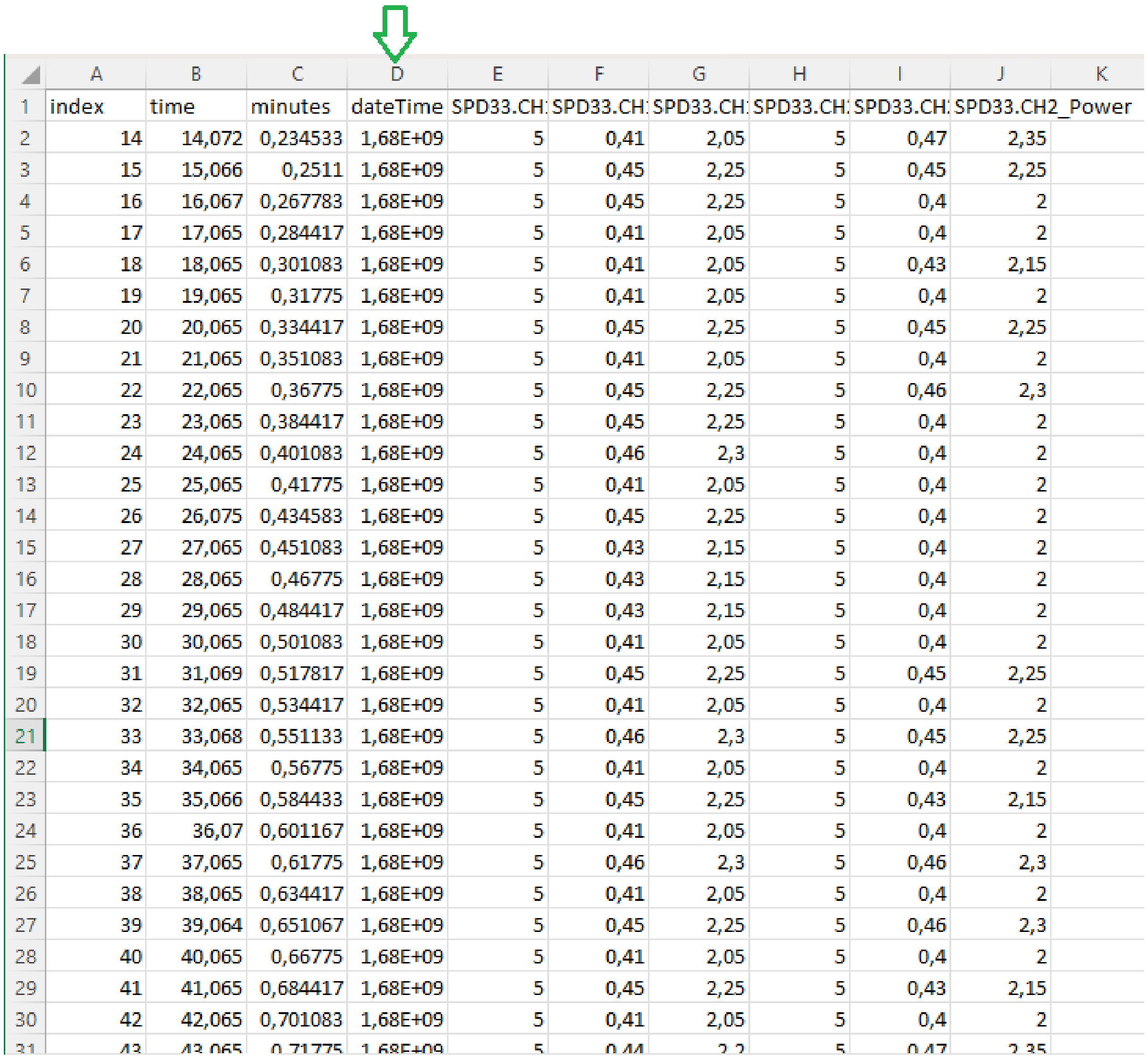
```
/home/dragosrazvan/Sustainable-API/bash
[Fri 28 Apr 09:51:51 BST 2023]
1[Fri 28 Apr 09:51:51 BST 2023] - before
1[Fri 28 Apr 09:52:57 BST 2023] - after
2[Fri 28 Apr 09:53:57 BST 2023] - before
2[Fri 28 Apr 09:55:03 BST 2023] - after
3[Fri 28 Apr 09:56:03 BST 2023] - before
3[Fri 28 Apr 09:57:08 BST 2023] - after
4[Fri 28 Apr 09:58:08 BST 2023] - before
4[Fri 28 Apr 09:59:14 BST 2023] - after
5[Fri 28 Apr 10:00:14 BST 2023] - before
5[Fri 28 Apr 10:01:19 BST 2023] - after
6[Fri 28 Apr 10:02:19 BST 2023] - before
6[Fri 28 Apr 10:03:24 BST 2023] - after
7[Fri 28 Apr 10:04:24 BST 2023] - before
7[Fri 28 Apr 10:05:29 BST 2023] - after
8[Fri 28 Apr 10:06:29 BST 2023] - before
8[Fri 28 Apr 10:07:34 BST 2023] - after
9[Fri 28 Apr 10:08:34 BST 2023] - before
9[Fri 28 Apr 10:09:39 BST 2023] - after
10[Fri 28 Apr 10:10:39 BST 2023] - before
10[Fri 28 Apr 10:11:45 BST 2023] - after
11[Fri 28 Apr 10:12:45 BST 2023] - before
11[Fri 28 Apr 10:13:51 BST 2023] - after
12[Fri 28 Apr 10:14:51 BST 2023] - before
12[Fri 28 Apr 10:15:56 BST 2023] - after
```

Figure 14. The log file generated by Bash for the Express server.

3. Take both dates related to a run and add 1 hour to convert the time from BST to CEST.

4. Convert the dates from step 3 to UNIX timestamp, since that is the format used in the logs from TestController, as seen in Figure 15.
   a. For this conversion, we have used an online Epoch converter[28].

[28] The official link for the Epoch Converter: https://www.epochconverter.com/



	A	B	C	D	E	F	G	H	I	J	K
1	index	time	minutes	dateTime	SPD33.CH	SPD33.CH	SPD33.CH	SPD33.CH	SPD33.CH	SPD33.CH2_Power	
2	14	14,072	0,234533	1,68E+09	5	0,41	2,05	5	0,47	2,35	
3	15	15,066	0,2511	1,68E+09	5	0,45	2,25	5	0,45	2,25	
4	16	16,067	0,267783	1,68E+09	5	0,45	2,25	5	0,4	2	
5	17	17,065	0,284417	1,68E+09	5	0,41	2,05	5	0,4	2	
6	18	18,065	0,301083	1,68E+09	5	0,41	2,05	5	0,43	2,15	
7	19	19,065	0,31775	1,68E+09	5	0,41	2,05	5	0,4	2	
8	20	20,065	0,334417	1,68E+09	5	0,45	2,25	5	0,45	2,25	
9	21	21,065	0,351083	1,68E+09	5	0,41	2,05	5	0,4	2	
10	22	22,065	0,36775	1,68E+09	5	0,45	2,25	5	0,46	2,3	
11	23	23,065	0,384417	1,68E+09	5	0,45	2,25	5	0,4	2	
12	24	24,065	0,401083	1,68E+09	5	0,46	2,3	5	0,4	2	
13	25	25,065	0,41775	1,68E+09	5	0,41	2,05	5	0,4	2	
14	26	26,075	0,434583	1,68E+09	5	0,45	2,25	5	0,4	2	
15	27	27,065	0,451083	1,68E+09	5	0,43	2,15	5	0,4	2	
16	28	28,065	0,46775	1,68E+09	5	0,43	2,15	5	0,4	2	
17	29	29,065	0,484417	1,68E+09	5	0,43	2,15	5	0,4	2	
18	30	30,065	0,501083	1,68E+09	5	0,41	2,05	5	0,4	2	
19	31	31,069	0,517817	1,68E+09	5	0,45	2,25	5	0,45	2,25	
20	32	32,065	0,534417	1,68E+09	5	0,41	2,05	5	0,4	2	
21	33	33,068	0,551133	1,68E+09	5	0,46	2,3	5	0,45	2,25	
22	34	34,065	0,56775	1,68E+09	5	0,41	2,05	5	0,4	2	
23	35	35,066	0,584433	1,68E+09	5	0,45	2,25	5	0,43	2,15	
24	36	36,07	0,601167	1,68E+09	5	0,41	2,05	5	0,4	2	
25	37	37,065	0,61775	1,68E+09	5	0,46	2,3	5	0,46	2,3	
26	38	38,065	0,634417	1,68E+09	5	0,41	2,05	5	0,4	2	
27	39	39,064	0,651067	1,68E+09	5	0,45	2,25	5	0,46	2,3	
28	40	40,065	0,66775	1,68E+09	5	0,41	2,05	5	0,4	2	
29	41	41,065	0,684417	1,68E+09	5	0,45	2,25	5	0,43	2,15	
30	42	42,065	0,701083	1,68E+09	5	0,41	2,05	5	0,4	2	
31	43	43,065	0,71775	1,68E+09	5	0,44	2,2	5	0,47	2,35	

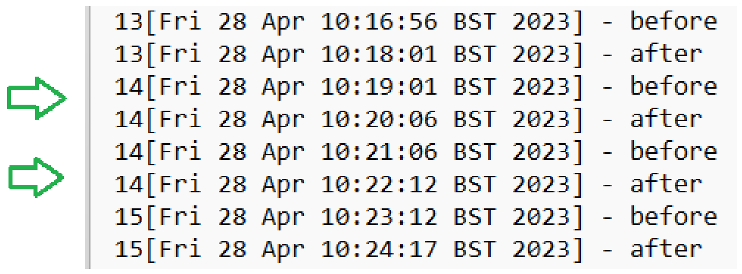
Figure 15. One log file exported from TestController, opened with Microsoft Excel.

5. In the results from the TestController, select all rows starting with the entry that has the matching "before date" calculated in step number 4 and up to (including) the one having the "after date" calculated in the same manner.

6. Create a new CSV file with the same headers as the ones from Figure 12 (i.e. index, time, minutes, dateTime, voltage, current, and power for both the client and the server), then copy and paste the preselected data from step 5.

7. Repeat the steps from 1 to 6 (but without creating the headers again), continuing with the next date from the Bash script log.

This was done for each framework, resulting in thirty-one CSV files per framework and payload, with a total of 248 files (i.e. 31 files * 2 payloads * 4 frameworks). The reason for ending up with 31 files instead of the 30 iterations that we have coded in our script is due to an extra measurement when reaching the 14th iteration. The reason for this extra measurement is unknown to us, as we have not implemented such logic in our code, and we have only noticed it while analyzing and preparing the data.



```
13[Fri 28 Apr 10:16:56 BST 2023] - before
13[Fri 28 Apr 10:18:01 BST 2023] - after
14[Fri 28 Apr 10:19:01 BST 2023] - before
14[Fri 28 Apr 10:20:06 BST 2023] - after
14[Fri 28 Apr 10:21:06 BST 2023] - before
14[Fri 28 Apr 10:22:12 BST 2023] - after
15[Fri 28 Apr 10:23:12 BST 2023] - before
15[Fri 28 Apr 10:24:17 BST 2023] - after
```

Figure 16. Anomaly of an extra run in our script execution.

However, given that this happened with all runs and not just one, we decided to leave it and take advantage of those additional readings.

5.2. Analysis

Once we were done with preparing the samples and the files for each framework, we proceeded with the statistical analysis and comparison by using Python, together with Jupyter Notebooks.

Our analysis started with importing all the required libraries (as pointed out in “3.9. Python and Jupyter Notebook”) and loading individual CSV files linked to a framework and payload into Pandas DataFrames. Afterward, we created a list containing the power-related values for each experiment. This list was then used to calculate the variance of the elements by using Numpy’s “np.var()” method, which takes the average of the squared deviations from the mean. We were then able to determine the number of samples needed by using the variance in Cochran’s formula (chapter “3.10.8. Cochran’s formula for sample size”) with a 98 confidence level. Given that in all cases the required sample size resulting from Cochran’s formula was much less than the data we had already collected by using the methodology described in “4.3.4. Automated script”, we concluded that no more measurements were necessary to conduct our analysis.

After establishing the required sample size, we performed the 3 normal distribution tests mentioned previously, namely Shapiro-Wilk, D'Agostino's K^2, and Anderson-Darling. All tests indicated that data is non-normally distributed, meaning that the data is skewed or not symmetric around the mean. This could be due to several factors, such as outliers due to energy spikes or a non-linear relationship between the variables being measured.

In our A/B testing, we conducted a comparison between pairs of 2 samples by subtracting the value of the lower sample from the higher one. Out of all 24 (between the client and the server and each payload, 6 permutations could be made) pairs of values and sample permutations, we found that in 18 pairs, the empirical p-value was less than 0.02. This indicates that in 75% of the pairs, we had sufficient evidence to reject the null hypothesis, suggesting that the difference between the samples is statistically significant and valid.

However, in 25% of the cases, the observed data showed less extreme differences than what would be expected. This implies that for some pairs, there was not enough evidence to reject the null hypothesis at a significance level of 0.02. In other words, for these cases, the empirical evidence did not provide strong support for rejecting the null hypothesis. This does not necessarily imply that there is no difference between the samples, but rather that the observed differences may be due to random chance or variability. The p-values associated with these cases were higher, indicating a higher likelihood of obtaining similar or less extreme differences between the samples. Although these observed values did not exhibit a significant statistical distinction, it is important to consider potential practical implications regarding resource consumption between the two samples. Further analysis and careful consideration of these practical implications are necessary to gain a comprehensive understanding of the A/B testing results. While statistical significance provides valuable insights, it is equally important to evaluate the findings in light of real-world context and domain knowledge.

The next step was reindexing all DataFrames, as the indexes for all but the first sample were sequential but not starting from 0. Another reason for reindexing the DataFrames was that some of the data points were unexpectedly skipped by the TestController. Next, we calculated the time duration by using the “len()” method, the mean power via “mean()”, and the area under the curve through both Numpy’s “np.trapz()” method, which takes the power values as the first parameter and the new index as the second, and by using the formula of average power multiplied by the time duration. In our case, both the new index and the time were equal, since the new index is incremented by 1 at every step and the logging time used for our measurements was 1 second.

Similarly to the list containing all the power values, we have created DataFrames containing all mean power values, time lengths, and the areas under the curve as the first column, and an index of the run (from 1 to 31) as the second column, so we knew which value was associated with which run.

Having all the data of each framework and payload, we have plotted them by using scatter plots, histograms, KDE plots, and box plots for better visualization. Finally, all the data was saved in separate files based on its utility or function: a CSV file for the mean power results, one for the times, another one for the areas under the curve, and the last one for all power values.

All the aforementioned files are then imported into another Notebook where we compare the frameworks based on their payload and whether we consider the client or the server as the point of interest. In this former Notebook, we again import all the necessary libraries for manipulating and displaying the data. We load the files and concatenate the values for all frameworks into one DataFrame, which includes the index of each run, then we add a new column with the framework name and assign colors for each experiment. Following that, to compare all the frameworks, we plotted multiple graphs with 4 boxplots showing all areas under the curve and the distribution of the data with the median and the outliers. Barplot graphs were used everywhere else: for the mean energy consumption, mean power dissipation, mean time duration, comparison of the results for the energy consumption approximation method, and payload comparison for each framework. All figures are found in the chapter “6. Results”.

6. Results

This chapter showcases the results of the data analysis, where, among others, we compare the energy consumption of both the server and client for each of the aforementioned payloads and frameworks. To do this, we have used a combination of box plots and bar plots that emphasize the difference between them.

6.1. Energy consumption

We mentioned that energy consumption was calculated in 2 ways. The reason for this was that we were not sure if the power supply and the TestController were reporting the average power per second (i.e. reporting the average power during the last interval of time) or reporting the instant power, as each method requires a different way of approximating the consumption.

However, after analyzing and processing the data calculated by both using the trapezoid method and the formula of average power multiplied by the time for a run, we observed that the order of the frameworks with respect to their energy consumption stays the same, as seen below in Figure 17, 18, 19, and 20. In other words, the ranking remains unchanged, as the errors from using a less optimal approximation strategy fade out in a large sample. The idea of employing both methods of approximation to see if there is a difference in the results came after an e-mail correspondence with Luis Cruz, where he argued that the error from using the less optimal strategy should not be significant if we have a substantial sample size. More details about this matter can be found in the “7.4. Methods for approximating the energy consumption” subchapter.

All graphs have the area under the curve calculated with the formula of average power multiplied by the time interval, except for the ones in figures 17, 18, 19, and 20, which contain both approximation methods.

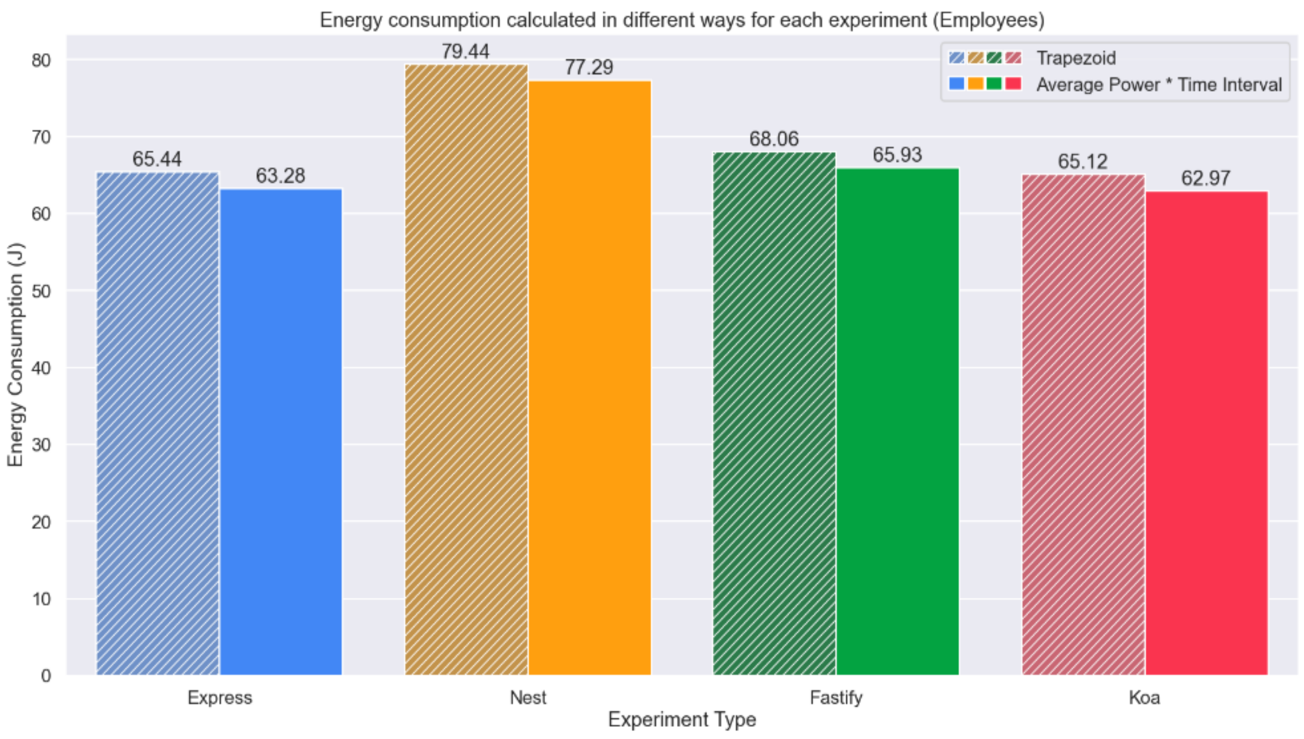


Figure 17. Bar plots of the area under the curve for all frameworks, calculated in both ways for the employees payload on the server.

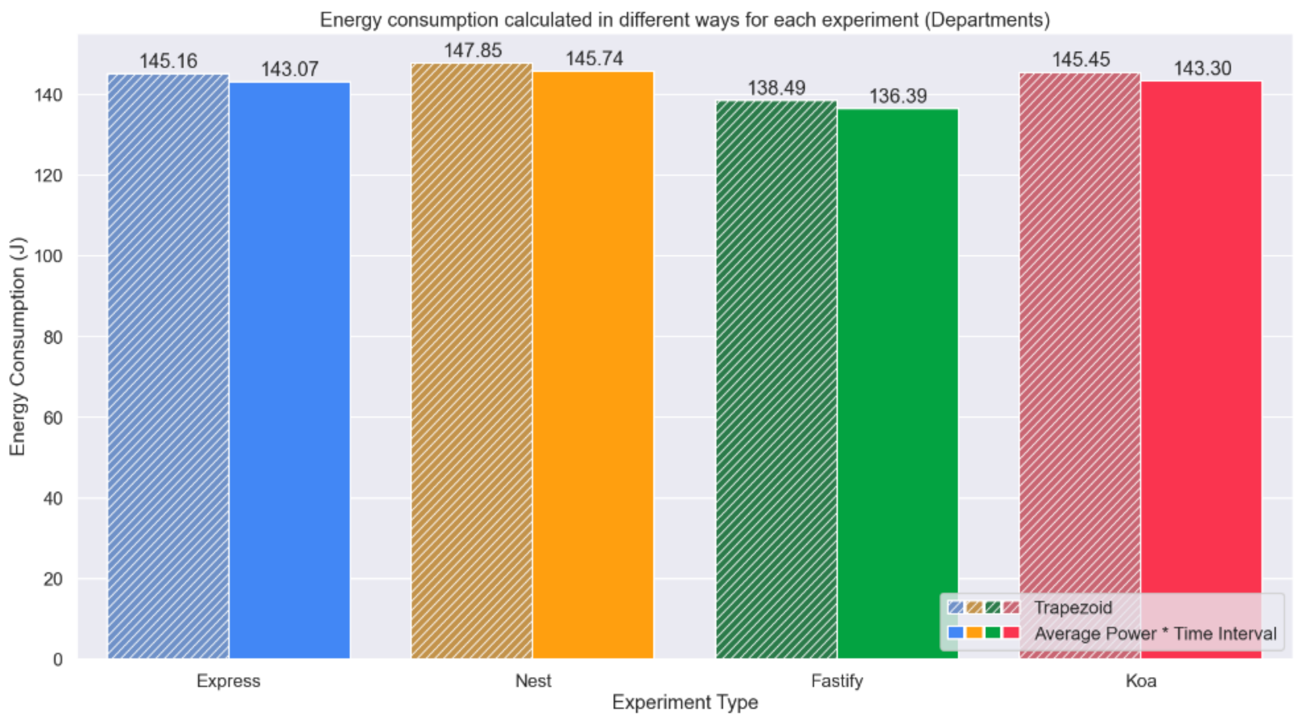


Figure 18. Bar plots of the area under the curve for all frameworks, calculated in both ways for the departments payload on the server.

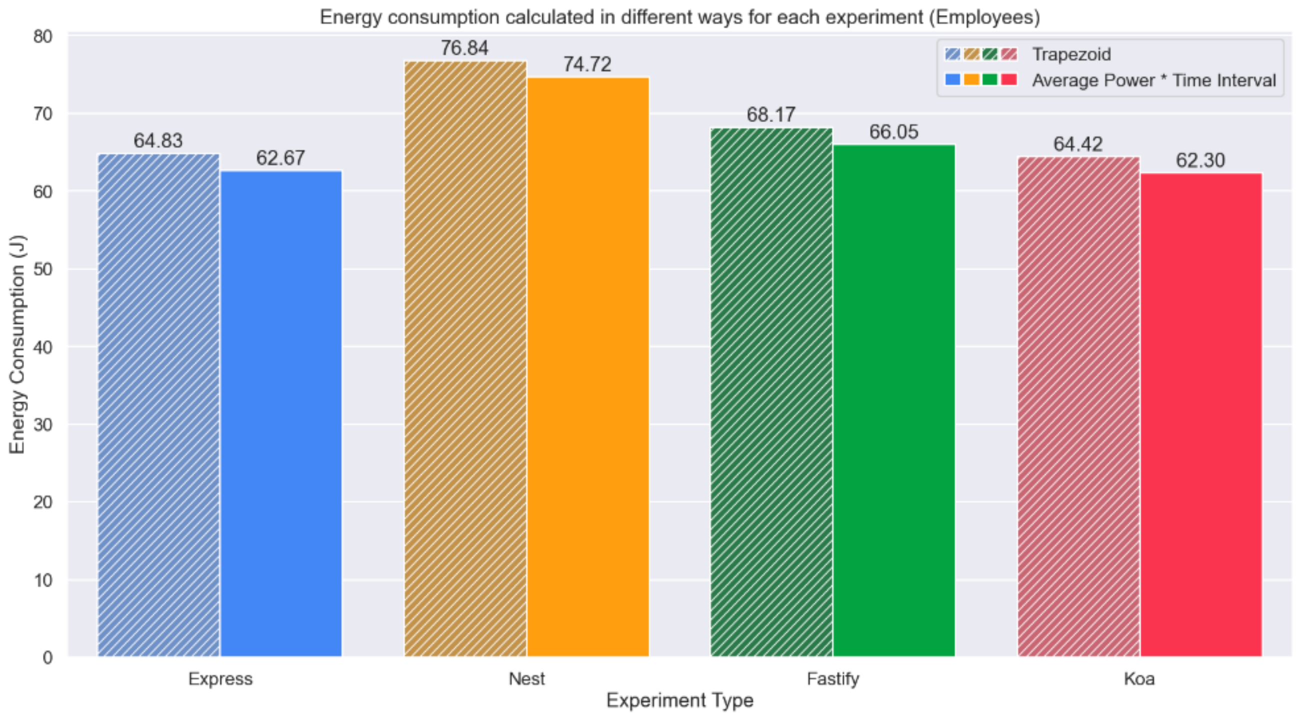


Figure 19. Bar plots of the area under the curve for all frameworks, calculated in both ways for the employees payload on the client.

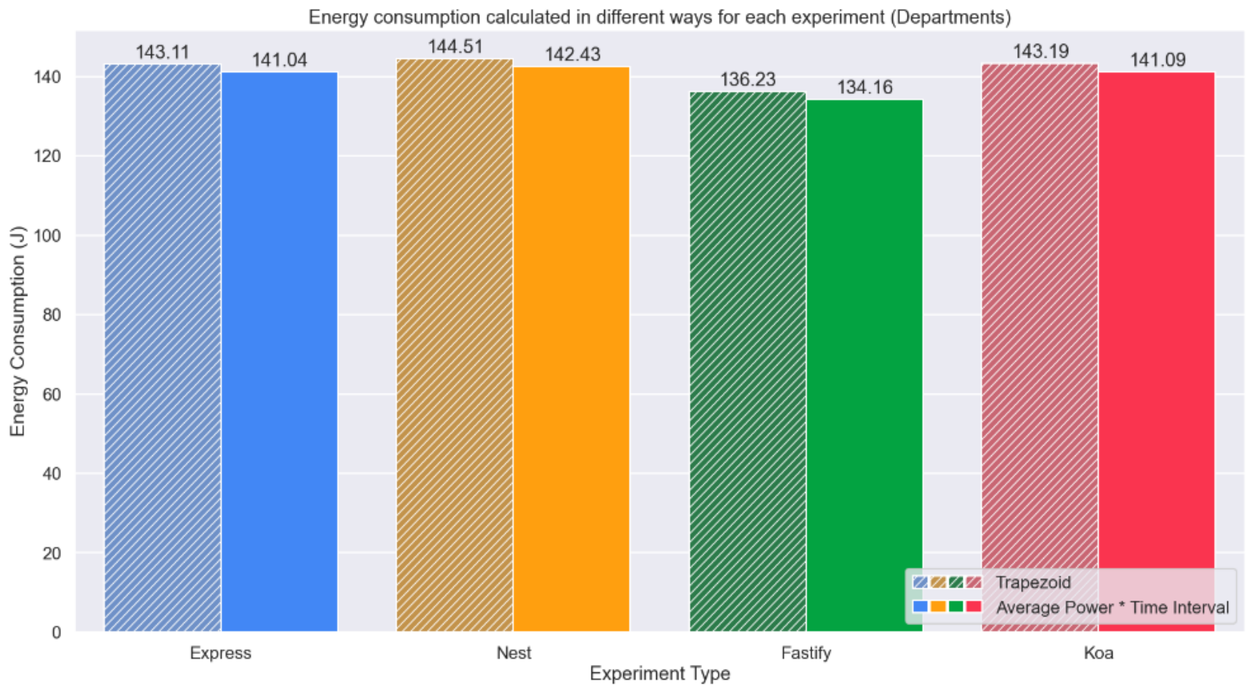


Figure 20. Bar plots of the area under the curve for all frameworks, calculated in both ways for the departments payload on the client.

Given that both Raspberry Pis were connected to the same power supply via different channels, the logs had the results for both the client and the server. The only adjustment that we had to make in our data analysis was to change the column name from channel 1 to channel 2 and then rerun all the Python code used for preparation and analysis for both cases. Hence, we are first presenting all the results associated with the server since all the API requests are processed here, followed by the data relating to the client, which only sends the requests and receives the response.

6.1.1. Server side

6.1.1.1. Employees

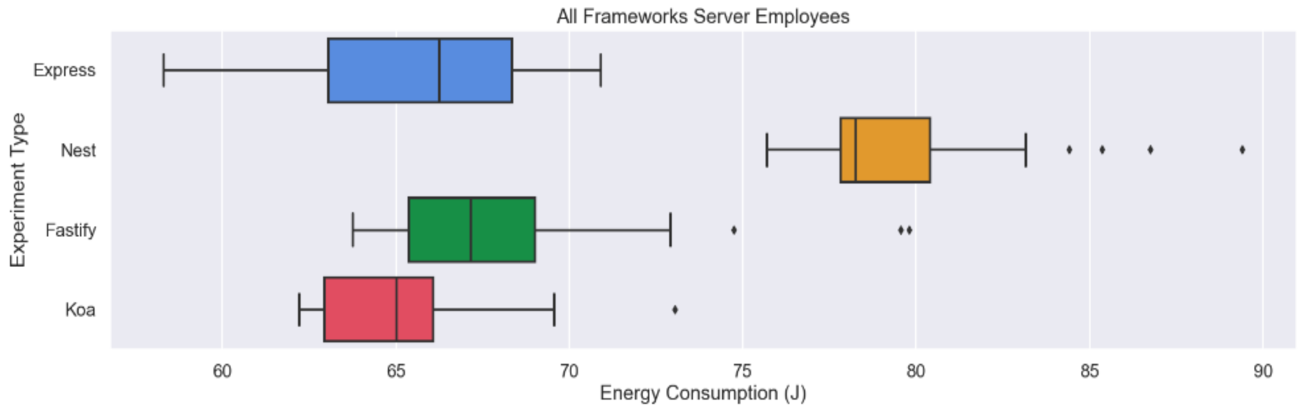


Figure 21. Box plots of the area under the curve for all frameworks, using the employees payload on the server side.

The above box plot provides a clear and concise summary of the energy consumption levels of the 4 frameworks: Express, Nest, Fastify, and Koa, on the server side. The plot's visualization displays the range, median, and quartiles of the energy consumption values for each framework, allowing for a quick comparison of their energy efficiency. For these results, we have applied the employees payload which is the bigger payload from the 2 that we used for this project.

From the plot, it is evident that Koa has the lowest energy consumption, with its energy consumption data being close in value to those of Express. This suggests that Koa and Express are both energy-efficient frameworks that are suitable for developers who prioritize sustainability, followed by Fastify. Nest, on the other hand, has the highest energy consumption levels among the 4 frameworks, indicating that it is the least energy-efficient framework.

6.1.1.2. Departments

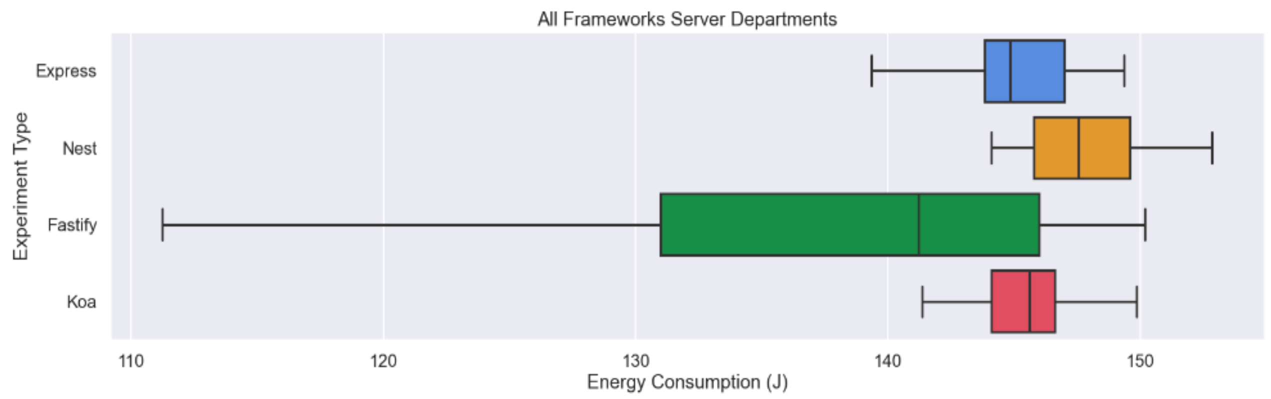


Figure 22. Box plots of the area under the curve for all frameworks, using the departments payload on the server side.

In this case, the results depicted above are based on the departments payload, which is the smallest of the 2. By analyzing the plot, it becomes clear that with this payload, Fastify has the lowest energy consumption levels, followed by Express. In contrast, Nest has the highest energy consumption levels among the 4 frameworks, implying that it is the least energy-efficient option, the same as with the bigger payload from Figure 21. However, as it can be seen, by using a smaller payload, Fastify is the one that consumes the least energy, while with a larger payload, Fastify is only in third place. At the same time, Koa stands only in third place with a small payload, but in the first position with a bigger payload in terms of the least consumption. Like Nest, Express is consistent and it maintains its second position both with a small payload and on a larger payload.

6.1.2. Client side

6.1.2.1. Employees

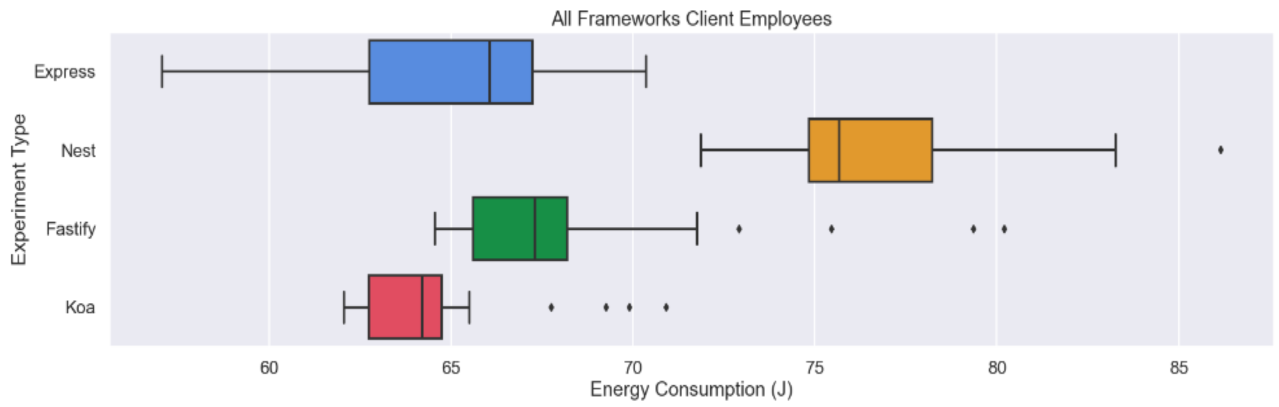


Figure 23. Box plots of the area under the curve for all frameworks, using the employees payload on the client side.

The box plot from Figure 23, presents the energy consumption of the same 4 frameworks but this time on the client side using the employees payload. Upon analyzing the plot, we see that Koa has the lowest energy consumption if comparing the median values. Furthermore, the variance of data points for Koa is smaller than that of Express. Fastify is the next in the ranking and once again, Nest has the highest energy consumption levels among the 4 frameworks, indicating that it is the least energy-efficient framework even on the client side.

6.1.2.2. Departments

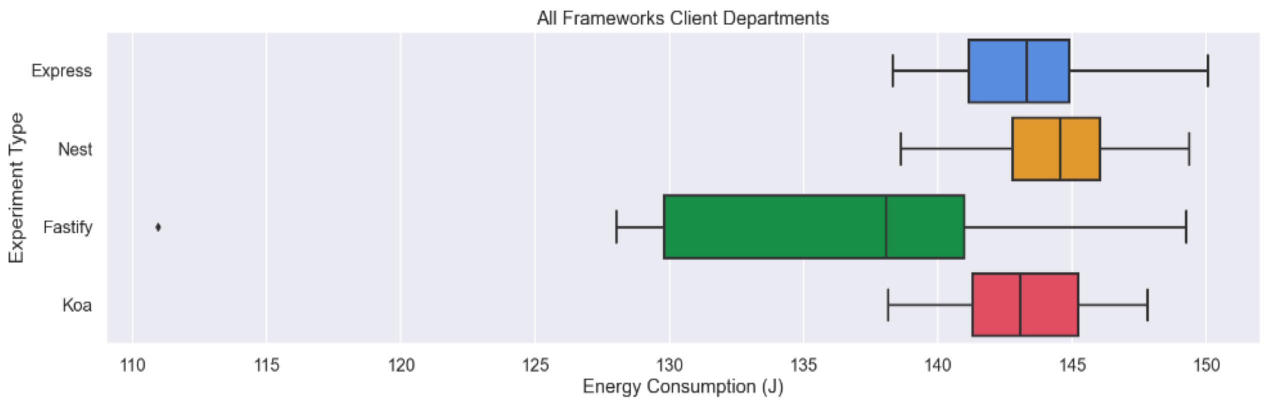


Figure 24. Box plots of the area under the curve for all frameworks, using the departments payload on the client side.

As you can see in the picture above, Fastify is the framework that consumes the least energy on the client side, as it was for the server with the same payload. An issue occurs when trying to find out which are the frameworks situated in the second and third position in terms of energy consumption because both Express and Koa seem to have almost the same starting point and even the median seems to be almost at the same level. However, one can observe that there is a slight difference and that Koa's median is indeed situated at a lower position on the scale compared to Express. At the same time, Nest keeps its last position as the framework that has the highest energy consumption between the 4 frameworks for both the client and server, regardless of the payload.

6.2. Frameworks payload comparisons on energy consumption

In this chapter, we conduct a comparative analysis of the energy consumption for different payloads but for the same framework. As mentioned earlier, we consider 2 distinct payloads: employees which are larger in size, and departments, which are smaller. These payloads are evaluated for both the client and server.

6.2.1. Server

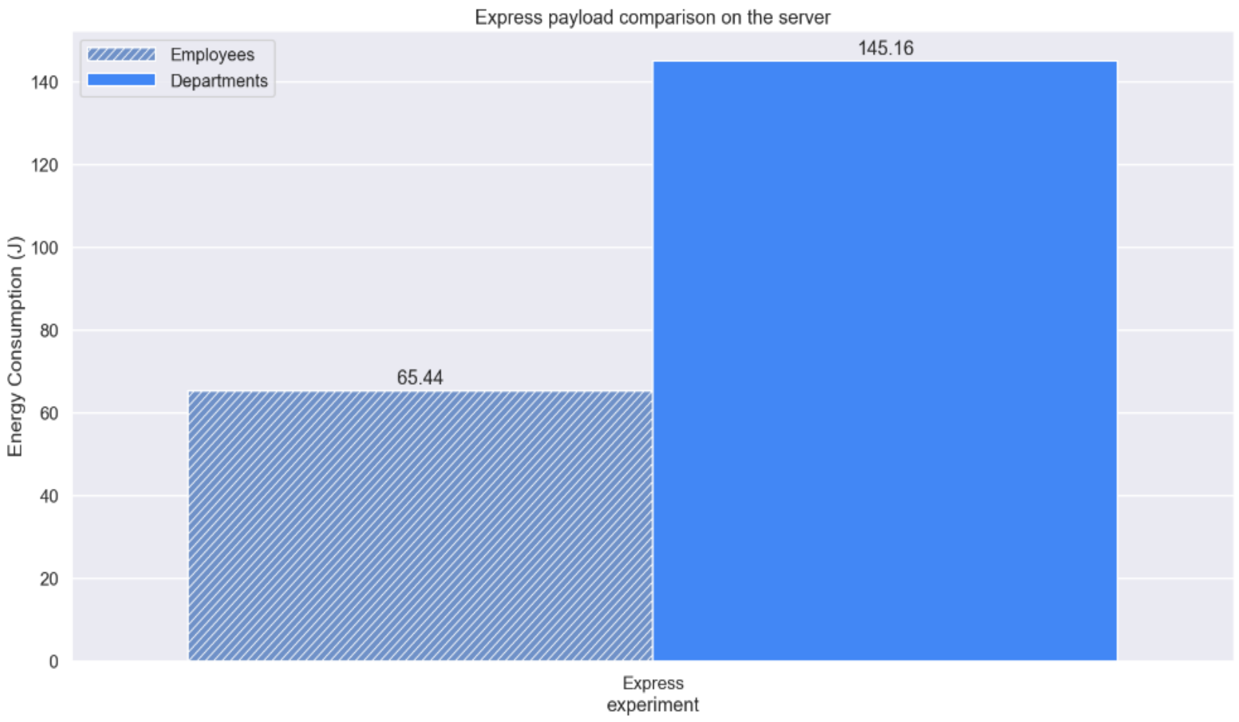


Figure 25. Energy consumption comparison between the employees and departments payloads using the Express framework on the server side.

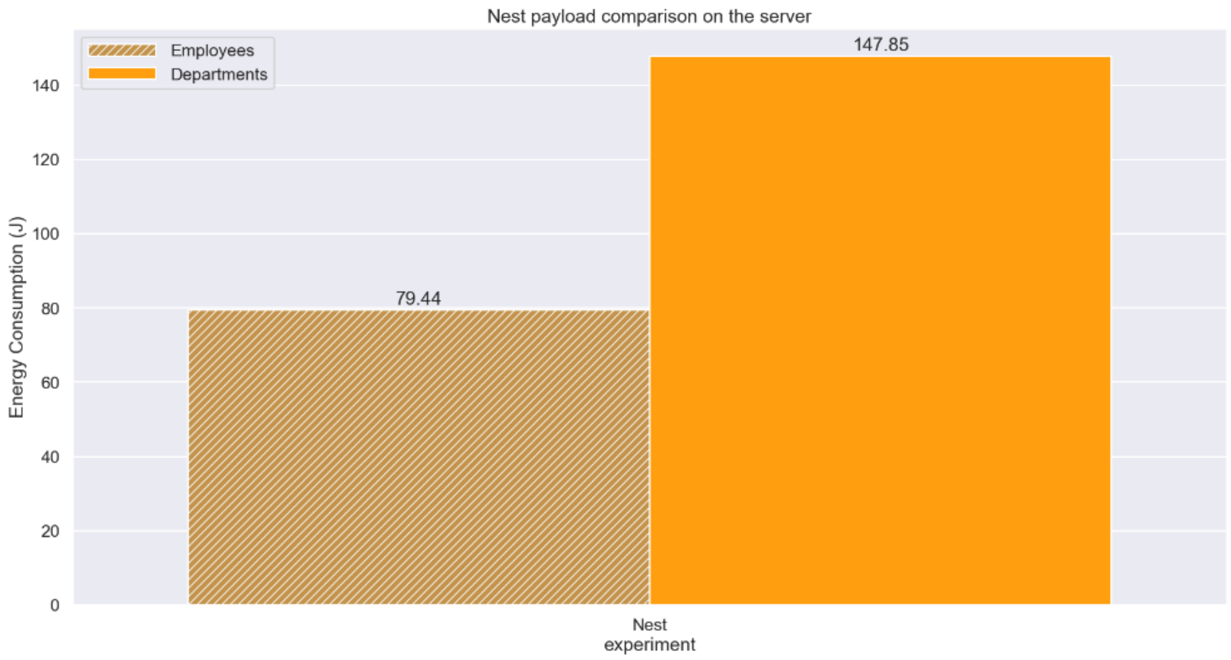


Figure 26. Energy consumption comparison between the employees and departments payloads using the Nest framework on the server side.

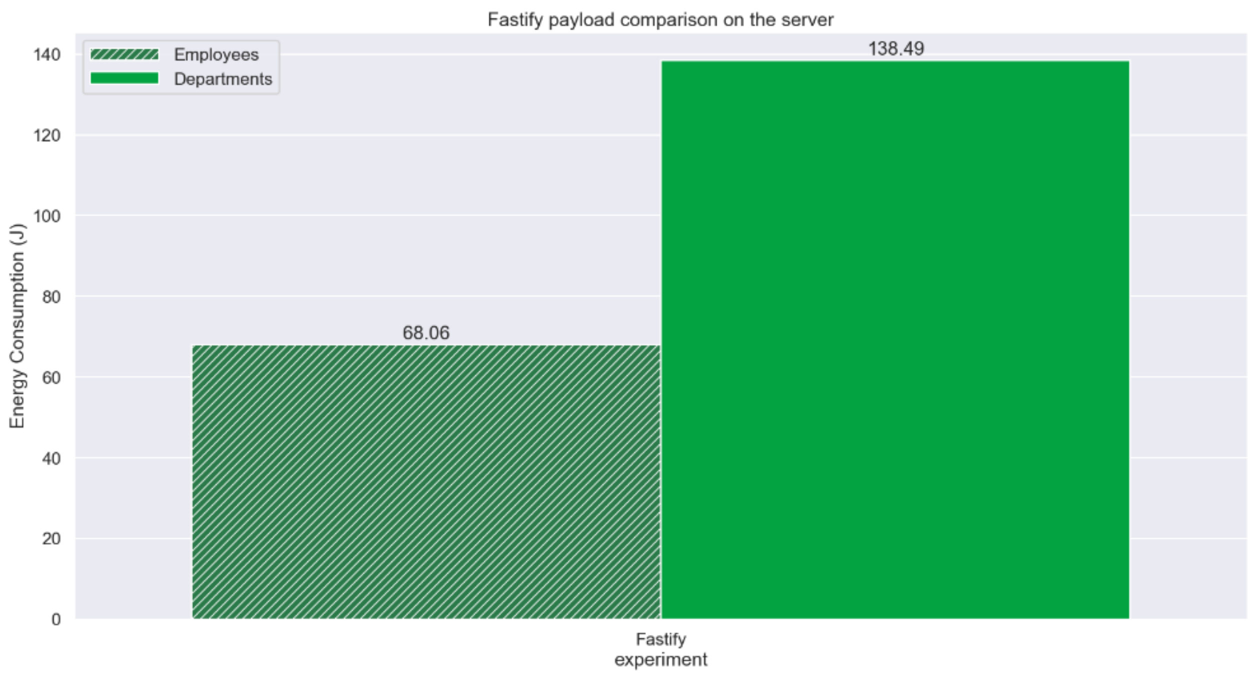


Figure 27. Energy consumption comparison between the employees and departments payloads using the Fastify framework on the server side.

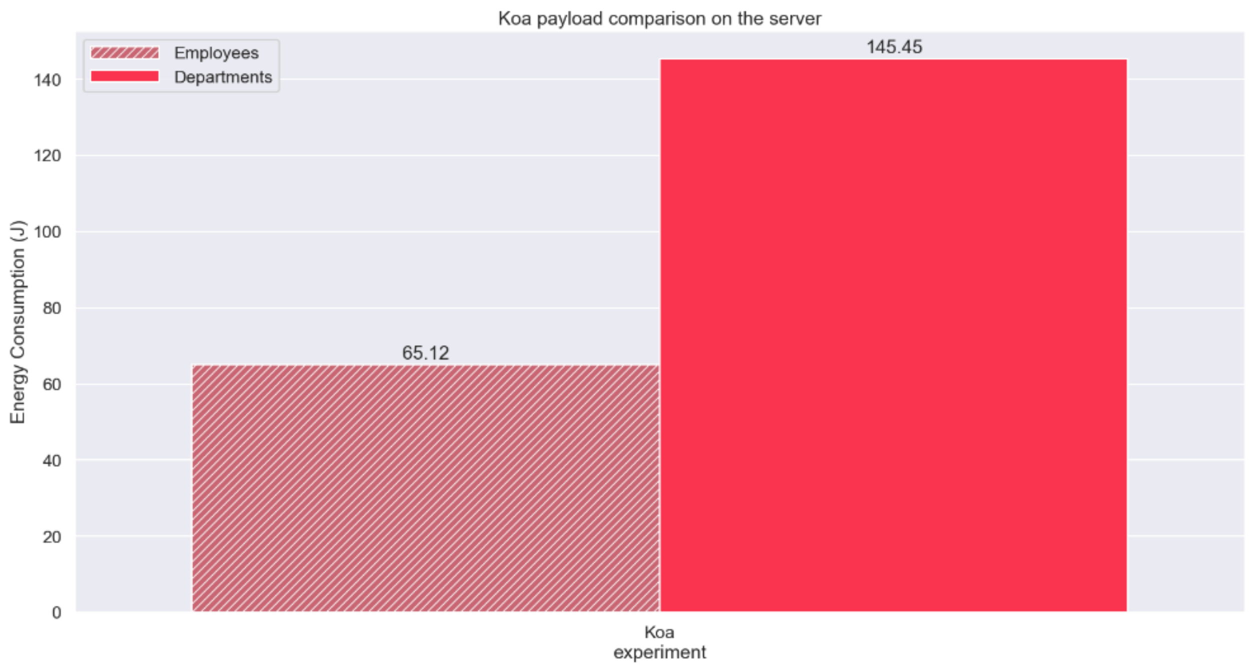


Figure 28. Energy consumption comparison between the employees and departments payloads using the Koa framework on the server side.

Surprisingly, it is consistently observed that the smaller payload (departments) exhibits higher energy consumption compared to the larger payload (employees) across all frameworks. This finding challenges the initial assumption that smaller payloads would result in lower energy consumption.

The unexpectedly higher energy consumption for the smaller payload remains unexplained, and on the same note, the execution time for the same entity was nearly double that of employees. These results defy the notion that smaller payloads inherently lead to reduced energy consumption, as their sizes would suggest.

Overall, these observations indicate that the distinction in energy consumption between a smaller and larger payload is contrary to what was expected, indicating that the power usage does not follow the anticipated pattern of increasing proportionally with the payload size, at least not in our experiments. These results highlight the need for further investigation into the underlying factors influencing energy consumption patterns and the interplay between payload size and energy efficiency across different frameworks.

6.2.2. Client

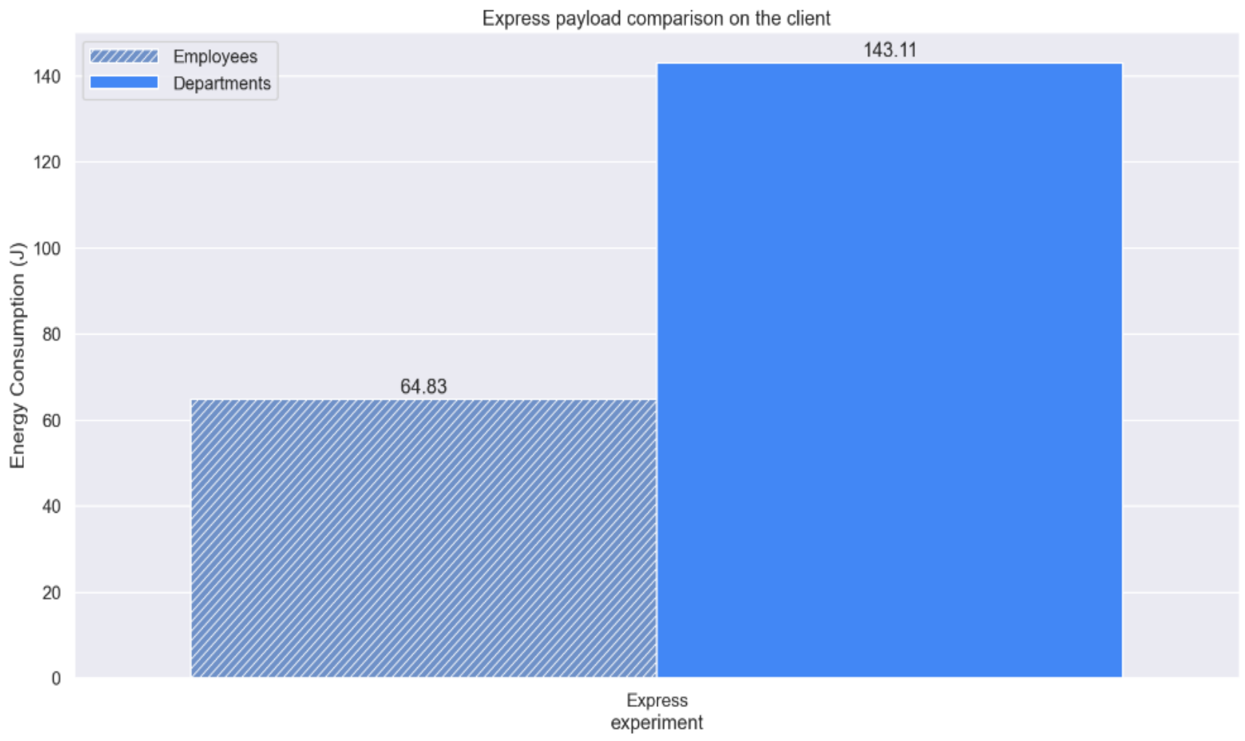


Figure 29. Energy consumption comparison between the employees and departments payloads using Express framework on the client side.

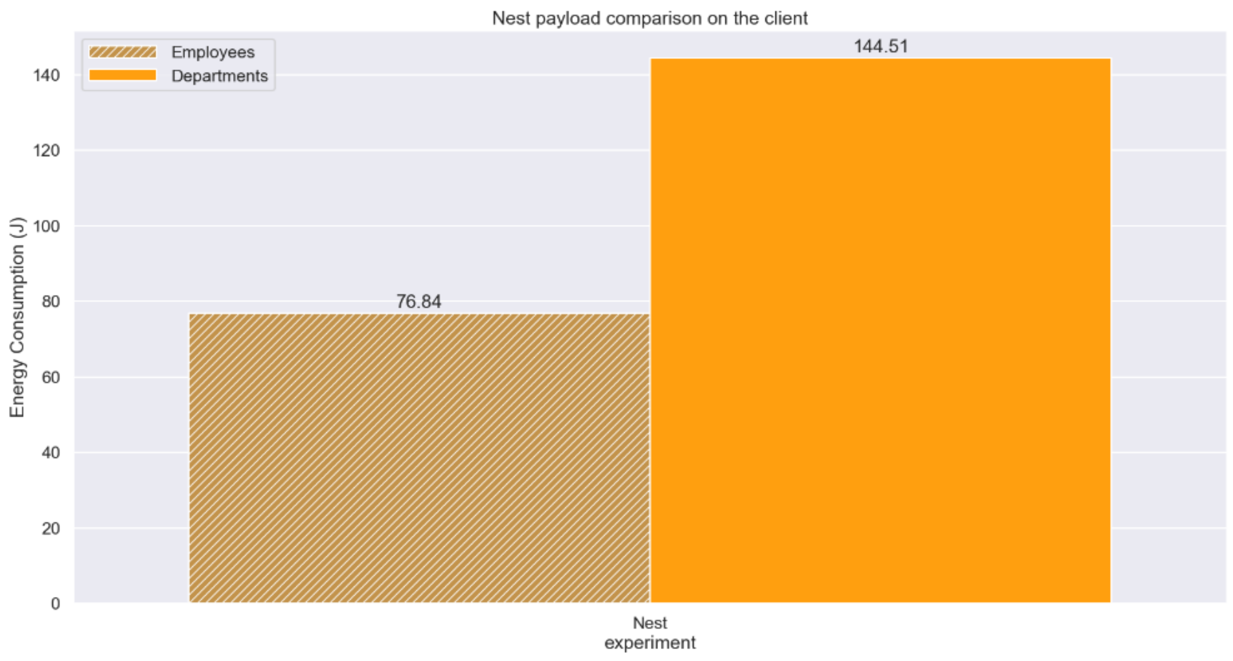


Figure 30. Energy consumption comparison between the employees and departments payloads using Nest framework on the client side.

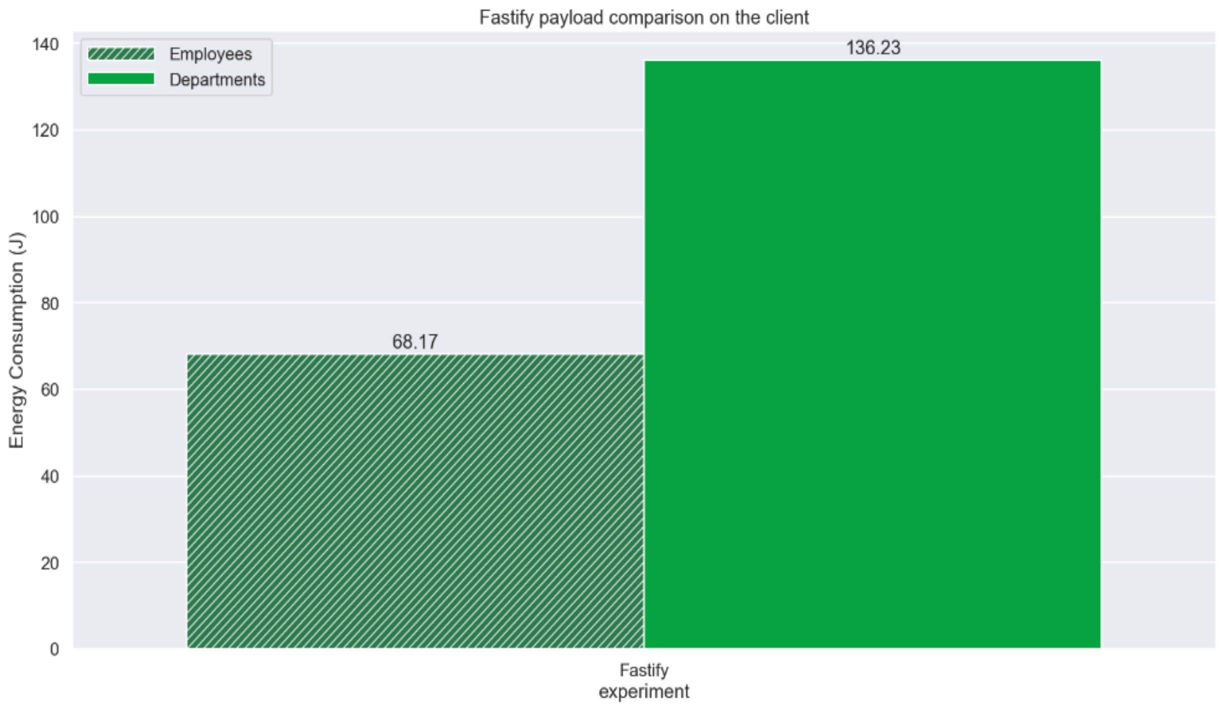


Figure 31. Energy consumption comparison between the employees and departments payloads using Fastify framework on the client side.

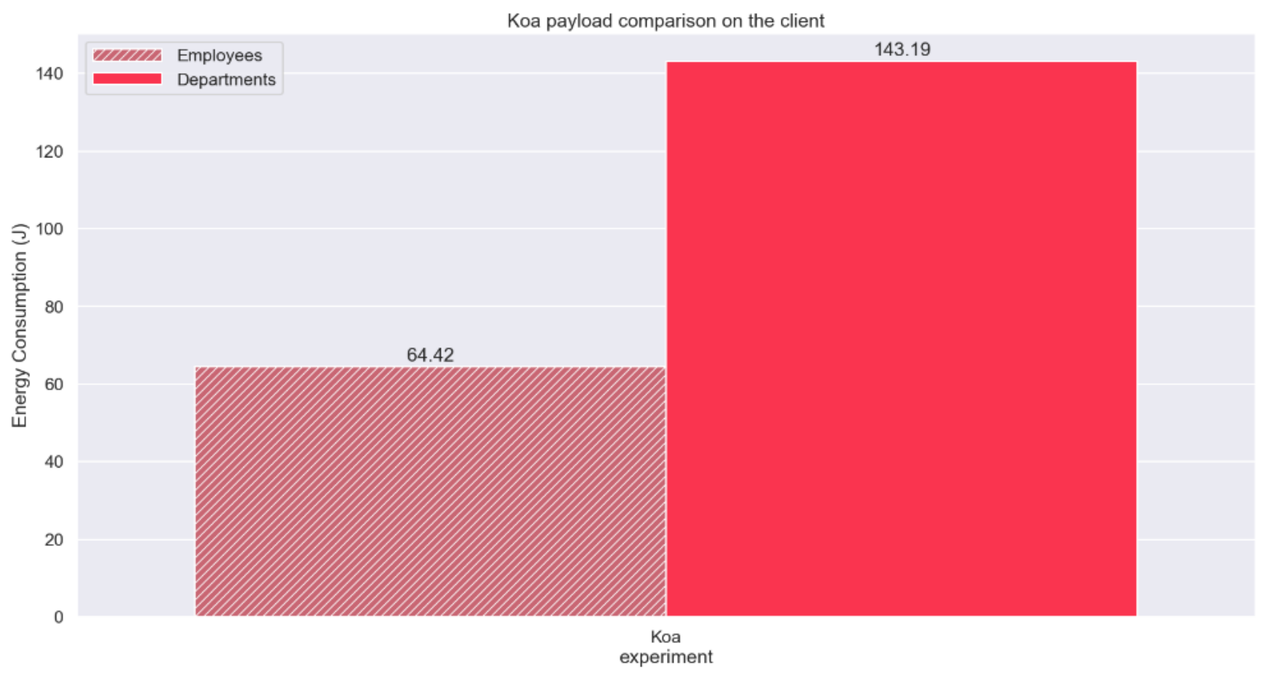


Figure 32. Energy consumption comparison between the employees and departments payloads using Koa framework on the client side.

In the same manner as in the server's examples, a consistent pattern also emerges here on the client side, wherein the smaller payload exhibits higher energy consumption compared to the larger payload across all frameworks.

6.3. Server and client comparison

In this chapter, we decided to compare the server and the client's energy consumption by using the same smaller and bigger payloads. As we wanted to go more in-depth with the comparison, we have also contrasted the server and client's mean energy consumption, mean power dissipation, and mean execution time.

6.3.1. Mean energy consumption

6.3.1.1. Employees

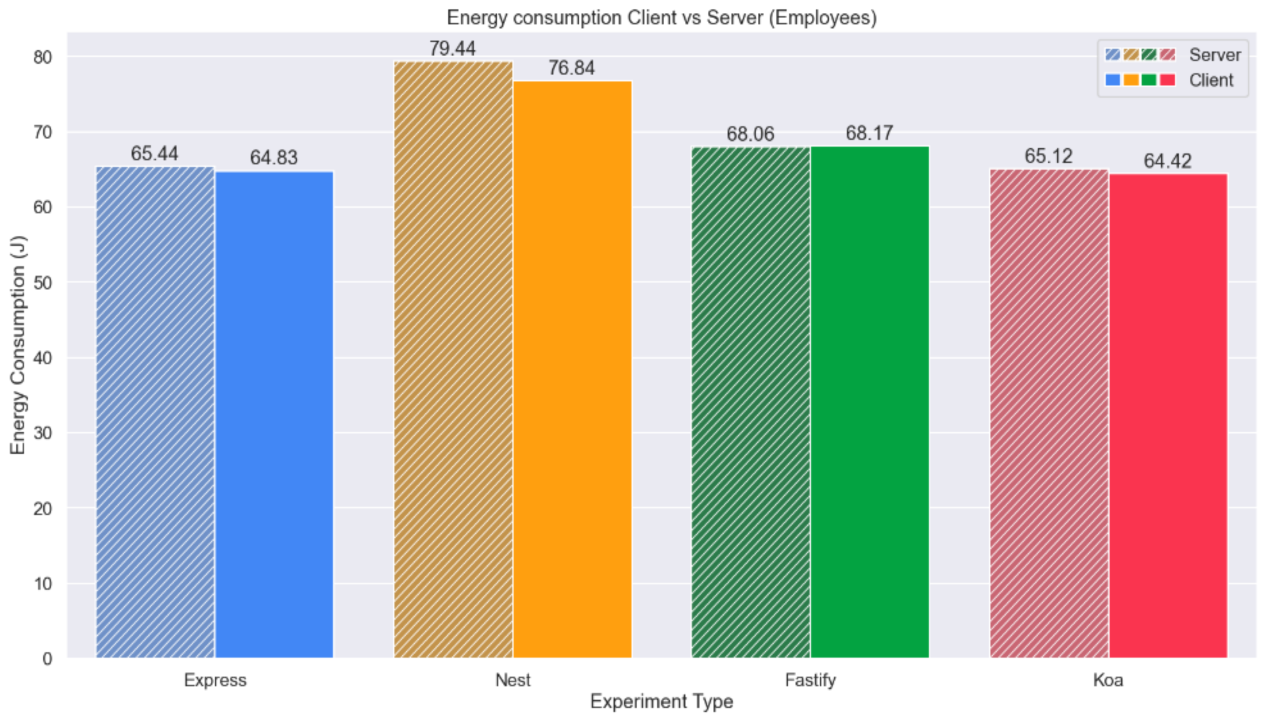


Figure 33. Mean energy consumption comparison between the server and the client using the employees payload.

The provided visual representation above offers a comparison between the energy consumption of all frameworks considering a larger payload. Notably, the comparison focuses on the distinction between the server and the client in terms of mean energy consumption. An important observation to be made is that the server exhibits higher energy consumption compared to the client in most cases. This finding aligns with expectations since the server is the one that handles and processes the requests from the client by running validations and establishing communication with the database, requiring additional resources and thus leading to increased energy usage. In contrast, clients have a relatively lower energy consumption due to their reduced computational requirements, having only the purpose of initiating requests and receiving responses. Surprisingly, there is one peculiar case with Fastify, where the client consumes slightly more energy than the server.

The ranking of the frameworks remains the same on both the server and the client side when using a bigger payload. Examining the results, the picture shows Koa which consistently demonstrates the lowest energy consumption, with Express closely behind, followed by Fastify, while Nest exhibits the highest energy consumption.

6.3.1.2. Departments

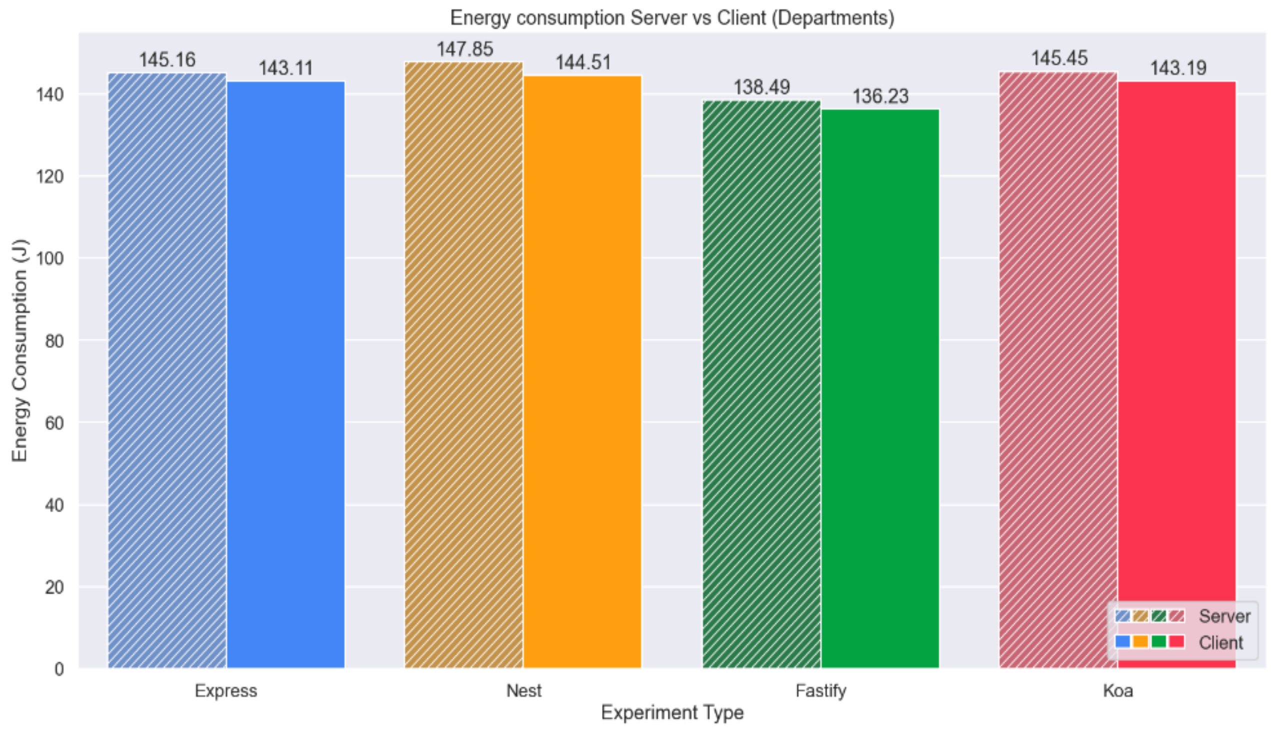


Figure 34. Mean energy consumption comparison between the server and the client using the departments payload.

The bar plot in Figure 34 depicts the comparison of the energy consumption among the same frameworks, this time considering a smaller payload. Once again, the server exhibits higher energy consumption compared to the client.

With the smaller payload, there are slight differences in the rankings between the server and the client. Despite Fastify maintaining the lowest energy consumption for both the server and the client and Nest maintaining the highest energy consumption, the rankings of Koa and Express vary. On the server side, Koa secures the second position, whereas Express ranks third in terms of energy consumption. Conversely, on the client side, Express takes the second spot while Koa follows closely in third place.

6.3.2. Mean power dissipation

6.3.2.1. Employees

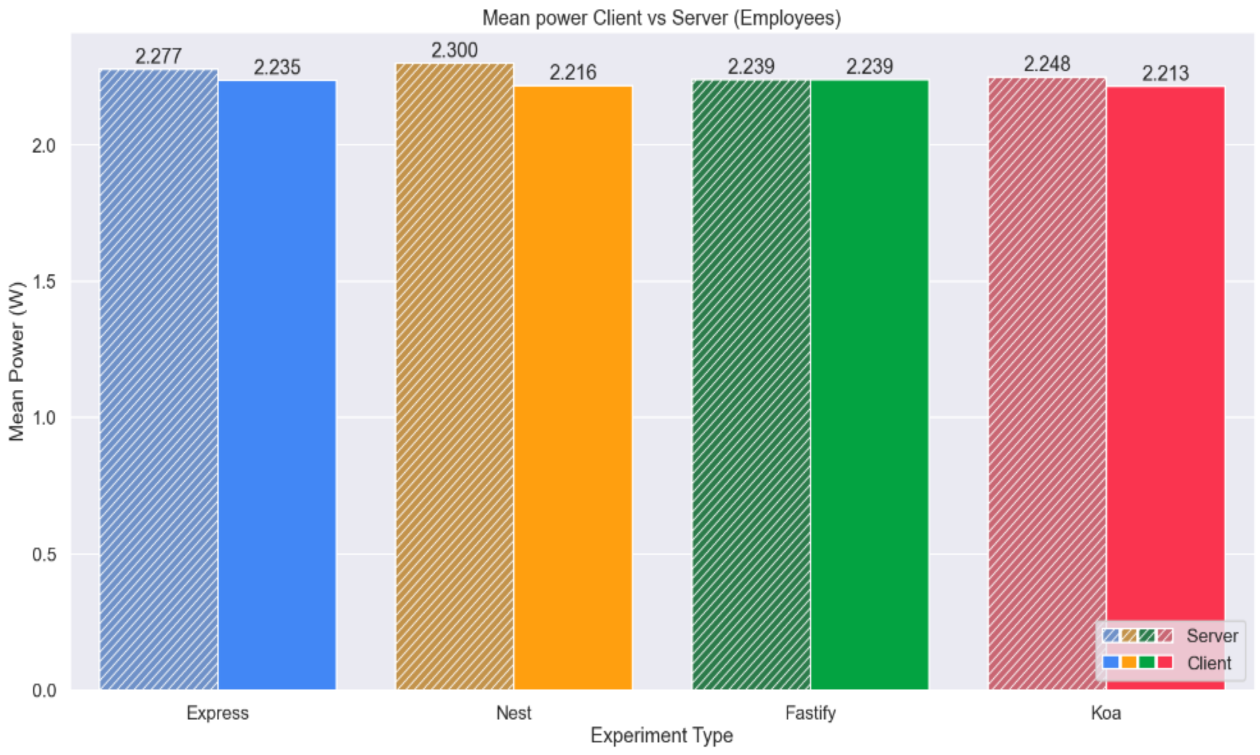


Figure 35. Mean power dissipation comparison between the server and the client using the employees payload.

The figure compares the average power dissipation. Similarly to energy consumption, the server generally demonstrates higher power dissipation compared to the client.

Upon closer examination, minor disparities in rankings between the server and the client are noticeable. Namely, Fastify stands out as the only framework with identical power dissipation values for both the client and server. Furthermore, Fastify exhibits the lowest power dissipation on the server, while simultaneously displaying the highest power dissipation on the client. Koa showcases notable efficiency yet again, ranking as the second-lowest power dissipation on the server and emerging at the top for the least power dissipation on the client side. Express maintains its consistent position as the third-ranked framework for power dissipation on both the server and the client. In contrast, Nest experiences the highest power dissipation on the server, but interestingly, it secures the second-lowest power dissipation on the client side.

6.3.2.2. Departments

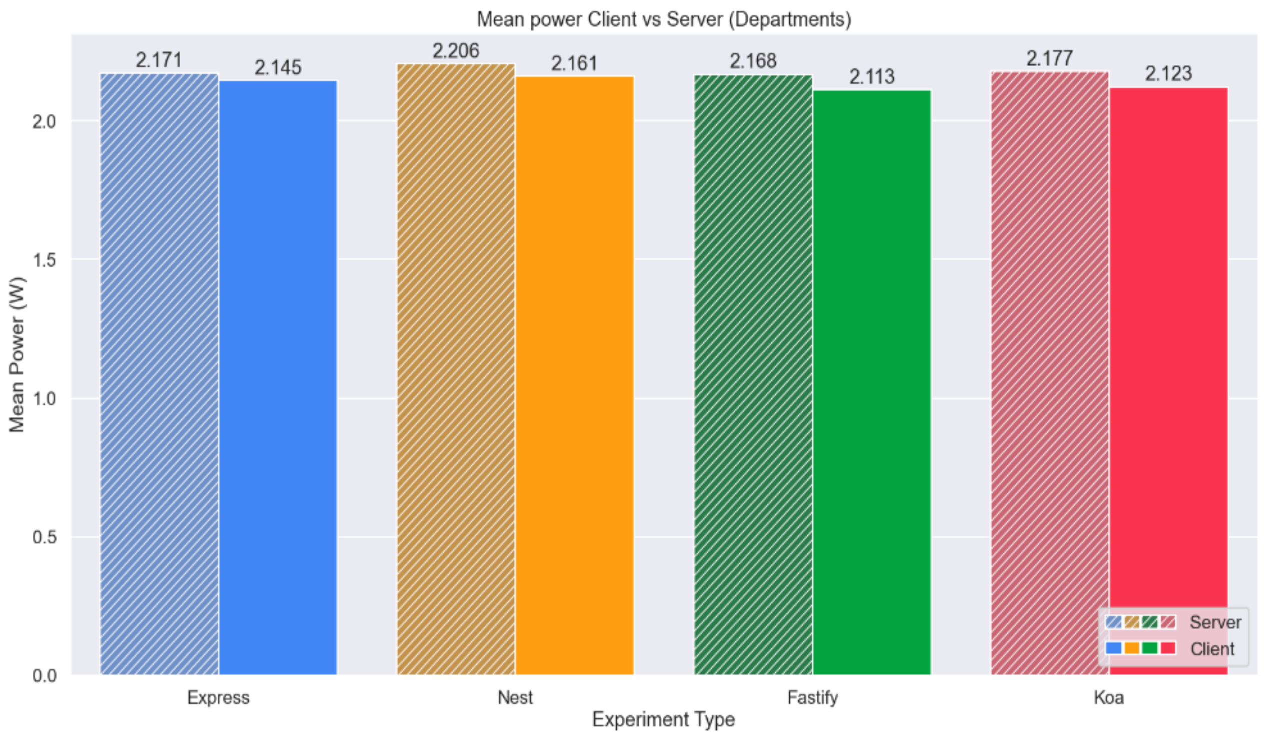


Figure 36. Mean power dissipation comparison between the server and the client using the departments payload.

When using the departments as payload, Fastify and Nest consistently maintain their respective positions on both the server and client sides. Fastify stands at the forefront, generally exhibiting the lowest power dissipation, while Nest remains at the bottom with the highest power dissipation.

In contrast, Koa and Express exchange their positions when transitioning between the server and client sides. On the server side, Express occupies the second position, while Koa ranks third in terms of power dissipation. However, on the client side, their positions are reversed, with Koa securing the second spot and Express following in third place in terms of power dissipation.

6.3.3. Mean execution time

6.3.3.1. Employees

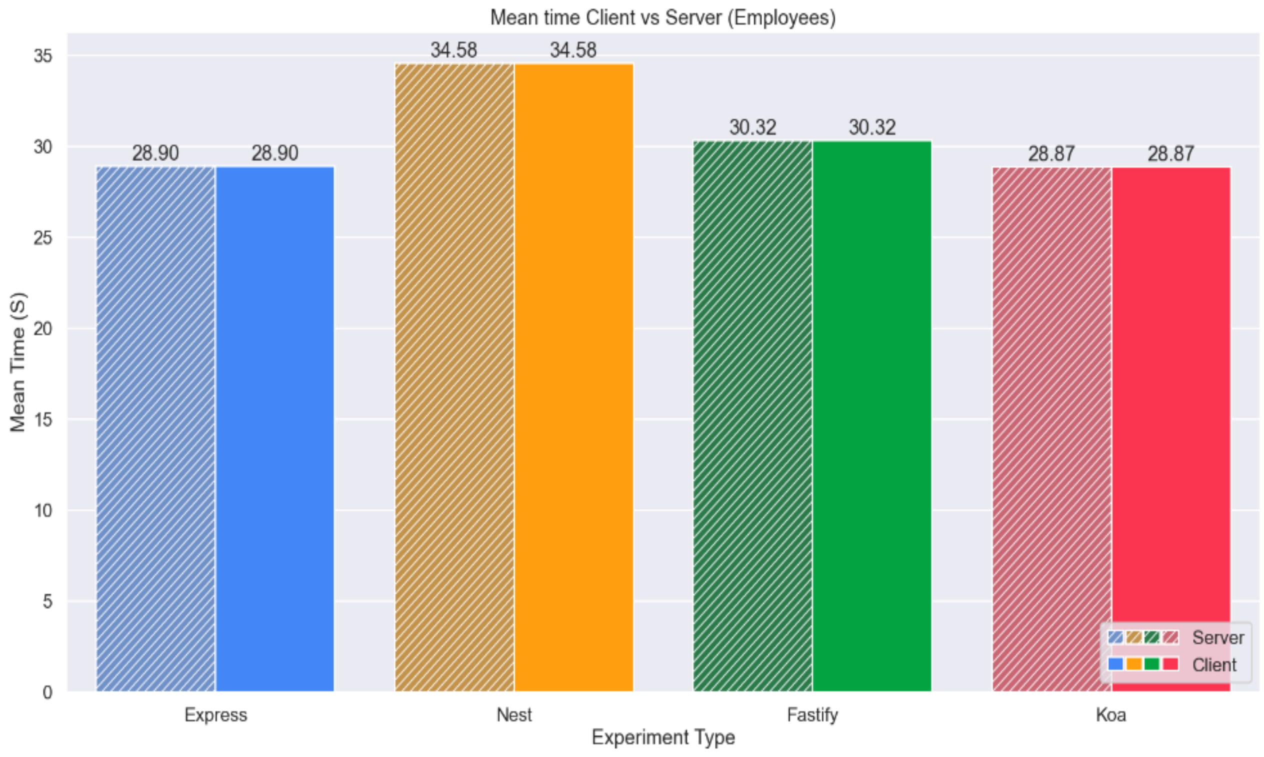


Figure 37. Mean execution time comparison between the server and the client using the employees payload.

When considering the mean execution time, in contrast to the measurements for energy consumption and power dissipation, the values and rankings for the server and client exhibit complete similarity.

As a result, Koa emerges as the frontrunner, showcasing the shortest mean execution time among the 4 frameworks. Following Koa, Express takes the second position, demonstrating a slightly longer mean execution time, while Fastify follows closely behind in third place. Nest, unfortunately, finds itself at the bottom of the rankings again, here with the longest mean execution time.

6.3.3.2. Departments

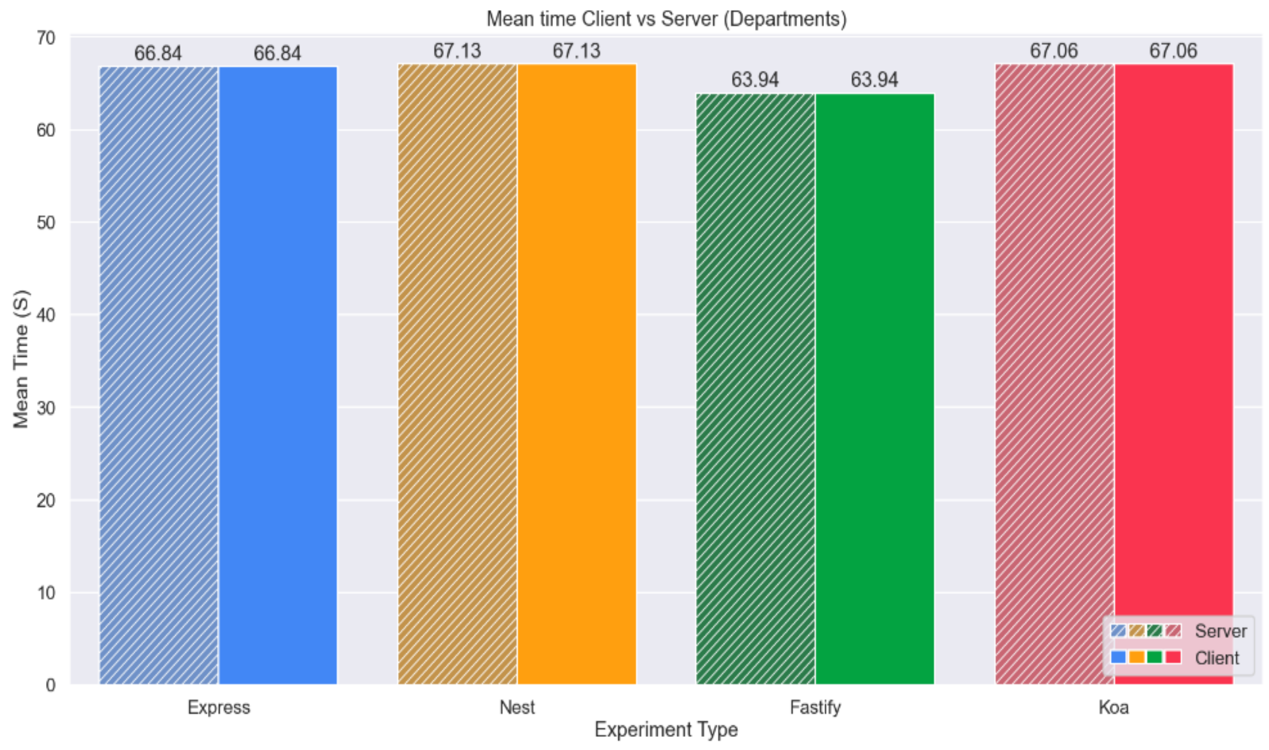


Figure 38. Mean execution time comparison between the server and the client using the departments payload.

As for a larger payload, the values and rankings for the server and client exhibit complete similarity when considering the mean execution time on the departments as well.

With the smaller payload, Fastify emerges as the frontrunner, demonstrating the shortest mean execution time among the 4 frameworks. Express maintains its position in second place, just as it did with a larger payload. Surprisingly, Koa is now positioned in third place, while once again, Nest finds itself at the bottom of the rankings, indicating the longest mean execution time among the frameworks.

7. Discussion

The chapter serves as a platform to delve into the findings of our research study, aiming to provide answers to some of the key questions raised and engage in a debate regarding the more interesting and unexpected results. By addressing both anticipated and unforeseen outcomes, we seek to gain a comprehensive understanding of the research findings and their broader implications. Through this process, we aim to contribute to the existing knowledge base and inspire further investigation into the subject matter.

7.1. Frameworks energy consumption comparison

In the section titled “6.1. Energy consumption”, NestJS demonstrated consistent behavior as the framework with the highest energy consumption. This aligns with our initial expectations when we began developing the APIs, considering that NestJS comes with many additional files and boilerplate code. According to the Nest documentation, the framework is built on TypeScript and provides strong typing capabilities, resulting in improved code quality. This influenced our decision to use TypeScript for building the API. In contrast, we used JavaScript for Express, Fastify, and Koa. Additionally, the documentation states that Nest utilizes other HTTP server frameworks such as Express or Fastify, making it more of an architectural framework that leverages the last 2 aforementioned frameworks. NestJS is a full-featured framework that follows modular, scalable, and maintainable architectural patterns inspired by Angular, providing predefined files, including controllers responsible for handling requests and returning responses to the client, modules used for organizing the application structure, and services designed to handle database queries, specifically intended for use by the controllers. Consequently, this arguably introduces unnecessary (at least for our context) overhead and complexity, which we believe may be more suitable for larger projects rather than for the 2 sets of APIs we developed.

In our examination of the efficiency evaluation of Node.js frameworks from (Demashov & Gosudarev, 2019), we noticed they utilize the same 4 frameworks that we employed, as well as others. Based on the conducted efficiency testing, Nest ranked lower compared to Fastify and Koa, which were identified as the 2 most efficient frameworks. This finding aligns with our research. However, in our experiment, Nest was not positioned ahead of Express as suggested by their results. As for the remaining 3 frameworks, we did not have specific expectations regarding their rankings, except for the understanding that Koa is an improved version of Express and that Fastify is more modern, aiming for speed and efficiency, hence expecting Express to be ranked below the other 2.

The results presented in the chapter “6.1. Energy consumption” provides evidence that Koa emerges as the most energy-efficient framework when utilizing a larger payload. In contrast, Fastify demonstrates superior efficiency when dealing with the smaller payload, with consistent results on both the client and server sides. Express.js, being one of the earliest and most popular backend Node.js frameworks, is renowned for its simplicity and ease of use, offering a lightweight framework with a vast ecosystem of middleware and plugins. However, Express can be considered less opinionated compared to other frameworks, which may require developers to make more decisions about the project structure and organization. Nevertheless, it consistently maintains the second or third position in terms of energy efficiency across various payload types, whether on the client or server side.

The Koa documentation indicates that it is developed by the same team responsible for Express, with the intention of creating an improved version of Express. Hence, we expected that Koa would exhibit greater energy efficiency compared to Express. However, our findings indicate that both frameworks are closely matched in terms of energy consumption. The same paper (Demashov & Gosudarev, 2019) shows that Koa has an advantage in terms of performance. However, it has a smaller core compared to Express.js and may require additional middleware or plugins for certain features.

Furthermore, the same research paper demonstrates that Fastify achieves the highest level of performance efficiency. Our research aligns with these findings as Fastify exhibits the lowest energy consumption, at least when handling a smaller payload. The Fastify documentation further supports our expectation that it focuses on speed and low overhead, leveraging modern JavaScript features to achieve high-performance benchmarks, being recognized for its exceptional performance and efficiency. However, Fastify may not be as feature-rich out-of-the-box compared to other frameworks, requiring developers to implement additional functionality through plugins or custom code.

In summary, although the specific intricacies of each framework are not completely known to us, our analysis has aimed to explain the discrepancies in energy consumption between them.

7.2. Adjustment of measurement methodology

To obtain more robust results and capture a comprehensive picture of the energy consumption patterns, we made an important adjustment to our data collection approach. Initially, measurements were conducted with a 1-minute break after each API call, intending to allow sufficient time for system stabilization and avoid potential interference between consecutive calls. However, we soon realized that this approach had limitations in capturing all the intricacies of power consumption during the actual execution of the calls. After analyzing these results, we noticed that the numbers were closer to the idle energy consumption, rather than seeing the actual consumption of our framework calls.

Because of this, we decided to introduce a modification to our methodology, and namely, we opted for a consecutive sequence of 30 API calls without any breaks, or rather introducing the break after all 30 consecutive calls were finalized. This adjustment proved to be crucial in enabling us to observe and analyze the finer details, including the presence of spikes and fluctuations that could have been missed under the previous setup due to the power supply logging. In situations where the power supply did not log the precise timing of each call, the 1-minute breaks would have obscured the visibility of potential energy spikes. By removing these breaks, we ensured that no valuable insights were missed, and we were able to capture a more comprehensive understanding of the energy consumption patterns.

The adoption of a consecutive sequence of 30 API calls not only improved the accuracy and granularity of our data but also enabled us to identify previously unnoticed spikes and variations in energy consumption. This adjustment has greatly enhanced the reliability and depth of our findings, allowing us to gain a more comprehensive understanding of the relationship between API calls and energy consumption. Besides, the absence of breaks between calls provided a more accurate representation of the energy consumed during the active execution of the API calls, as it closely mirrored real-world scenarios where requests are constantly stressing the server rather than having spread out or scattered clients making requests.

7.3. Server and client identical mean-time execution

In our experimental setup involving a single client and a single server in a RESTful API scenario, the observed synchronization of timing between the client and server can be attributed to the fundamental nature of their interaction. When a client sends a request to the server, it enters a state of waiting, allowing the server to process the request and generate a response. As a result, the client's overall duration is closely tied to the server's processing time. During the waiting period, the client remains idle until it receives a response from the server. This synchronization of timing ensures that the client and server appear to finish their tasks concurrently. The client's waiting time aligns with the server's processing time, leading to a perceived equal duration for both.

However, it's important to acknowledge that this synchronization is specific to our experimental setup with a single client and server. In real-world scenarios where multiple clients concurrently request information from the same server, the dynamics can change significantly. With concurrent or parallel requests, the server's workload increases, potentially leading to variations in processing time and energy consumption.

In situations where multiple clients are making concurrent requests, factors such as queuing, resource contention, and parallel processing come into play. These factors can introduce variations in the timing and energy consumption patterns among different clients and servers. As a result, the notion of the client and server finishing their tasks in the same amount of time may no longer hold.

Therefore, it's essential to consider the specific context and characteristics of the client-server interaction when evaluating the timing and energy consumption aspects. While our experiments with a single client and server demonstrate synchronized timing, real-world scenarios with multiple clients can yield different results, potentially leading to variations in energy consumption and performance.

7.4. Methods for approximating the energy consumption

In the initial phase of our experiment, we made the decision to use the trapezoid method for calculating energy consumption which considers the area under the power consumption curve.

We knew that this method is used when dealing with varying loads of consumption, where the energy consumption is not constant over time.

However, during our analysis, we became aware of another prevalent approach for calculating energy consumption, which involves using the formula of average power multiplied by the time interval (delta T). However, this second approach assumes a constant power consumption rate throughout the duration of the measurement.

Due to our lack of clarity about which one is applicable in our context of adopting both the power supply and the TestController piece of software, we were curious to investigate whether employing this alternative method would yield any notable differences in our findings. To address this, we decided to conduct a comparative analysis of the results obtained from both the trapezoid method and the average power multiplied by the time, which in the end, showed that the ranking of the frameworks remained consistent when sorted based on the energy consumption.

Additionally, we sought the expert opinion of Luis Cruz, who confirmed that when dealing with a large amount of data, the specific method of approximation should become less critical (which we demonstrated in the "6.1. Energy consumption" chapter). With a substantial volume of data points, the overall trend and patterns in energy consumption become more apparent, overshadowing any minor discrepancies that may arise from different approximation methods. This insight provided further reassurance that our focus should primarily be on the analysis and interpretation of the data, rather than emphasizing the choice of approximation method.

On the whole, our exploration of different methods of approximation and the endorsement from Luis Cruz emphasized the importance of considering the overall dataset and its characteristics. While the choice of approximation method can impact the precise numerical values, the general findings and trends in energy consumption are likely to remain consistent and meaningful when dealing with a large volume of data.

7.5. Payload comparison

To enhance the realism and broaden the scope of our study, we made the decision to incorporate 2 distinct payloads into our experiment. As discussed in chapter "4.3.2. Database", we selected these payloads from our database where we identified the employees entity as our larger payload, characterized by 6 fields, while the departments entity served as our smaller payload, featuring only 2 fields and a significantly smaller number of rows. Additionally, we developed a dedicated RESTful API to facilitate operations such as retrieving all rows, fetching a single row by id, creating new entities, updating existing entities, and deleting entities.

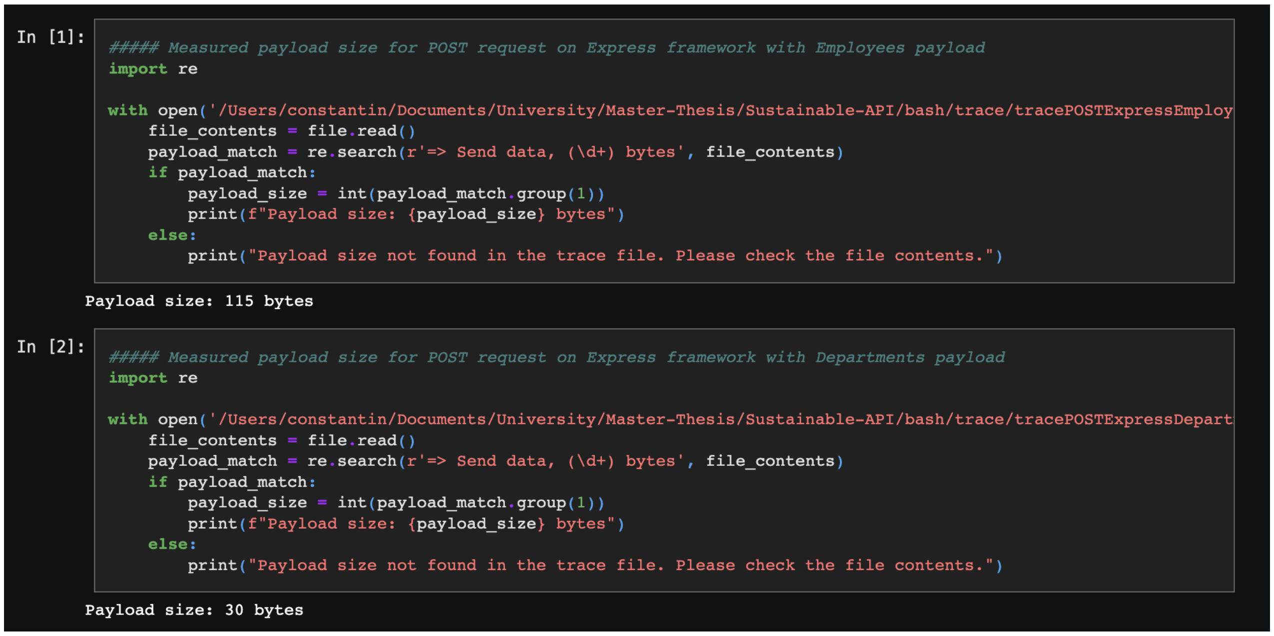
However, upon conducting the energy consumption measurements on these 2 payloads, we made an intriguing observation. The API requests directed towards the smaller payload, the department entity, not only exhibited a duration twice as long as those for the larger payload but also demonstrated sometimes a more than twofold increase in energy consumption, as outlined in the chapter "6.2. Frameworks payload comparisons on energy consumption". Remarkably, this trend of the smaller payload consuming more energy than the larger payload persisted across all 4 frameworks investigated in our research, indicating a consistent pattern rather than a mere coincidence. Furthermore, it is worth noting that we conducted multiple runs and tests (aside from the ones already mentioned), all of which yielded the same unexpected outcome.

Due to our initial assumptions, we expected the larger payload, which consisted of employees, to consume more energy. However, we currently lack an explanation as to why this occurred. As previously stated, additional investigation is necessary in order to provide a comprehensive explanation for the occurrence.

To verify our initial expectations, we attempted to measure the payload size for each entity by making requests to the APIs. To accomplish this, we utilized curl once again, employing the "--trace-ascii" option. Below is an example of a curl request used to obtain the payload size for a GET request on the Express framework, focusing on the employees payload:

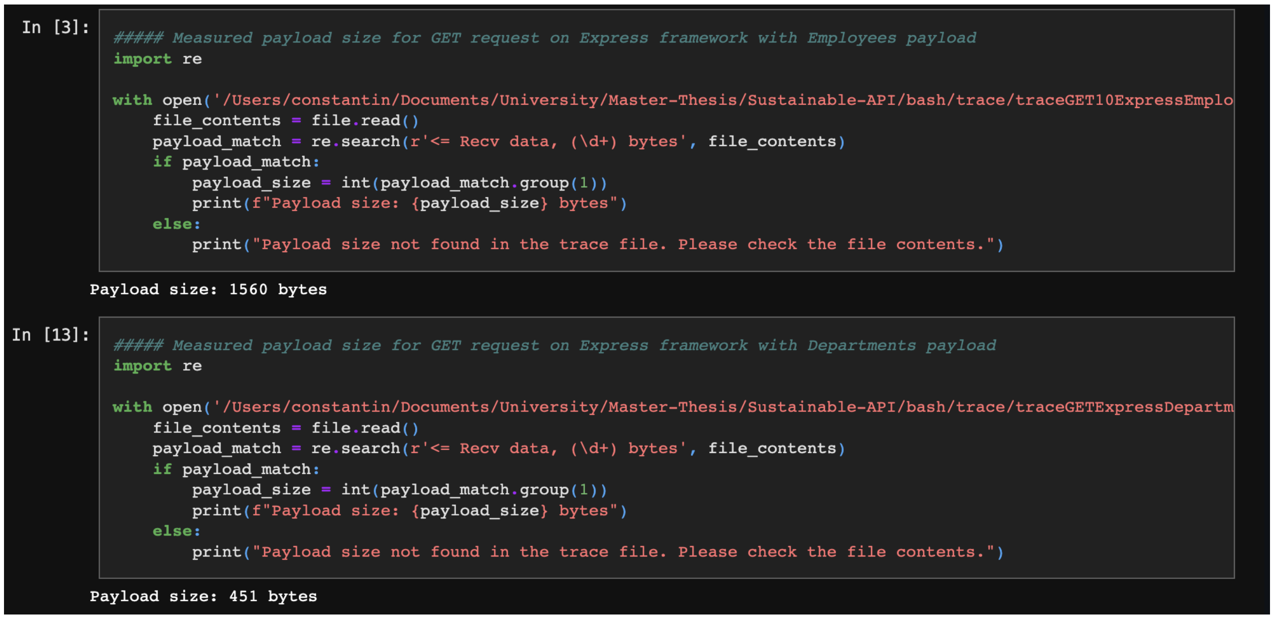
```
curl -H Content-Type:application/json --request GET --trace-ascii trace/trace.txt
http://localhost:3000/api/employees
```

Subsequently, we used Python and Jupyter Notebook to extract the data from the generated text file. This enabled us to conduct a comparative analysis of the payload sizes between the 2 entities. In accordance with our initial assumptions, we can demonstrate that the employee payload is indeed larger, as depicted in the accompanying images below, showing all 4 frameworks.



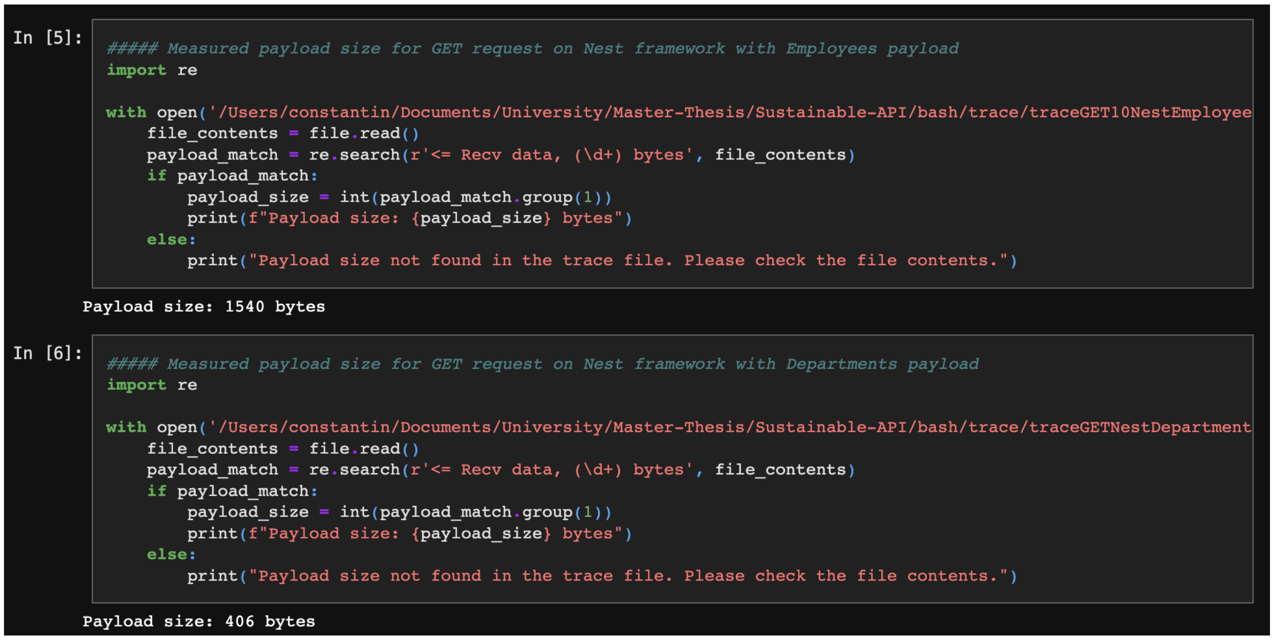
```
In [1]:
##### Measured payload size for POST request on Express framework with Employees payload
import re

with open('/Users/constantin/Documents/University/Master-Thesis/Sustainable-API/bash/trace/tracePOSTExpressEmploy
    file_contents = file.read()
    payload_match = re.search(r'=> Send data, (\d+) bytes', file_contents)
    if payload_match:
        payload_size = int(payload_match.group(1))
        print(f"Payload size: {payload_size} bytes")
    else:
        print("Payload size not found in the trace file. Please check the file contents.")

Payload size: 115 bytes

In [2]:
##### Measured payload size for POST request on Express framework with Departments payload
import re

with open('/Users/constantin/Documents/University/Master-Thesis/Sustainable-API/bash/trace/tracePOSTExpressDepart
    file_contents = file.read()
    payload_match = re.search(r'=> Send data, (\d+) bytes', file_contents)
    if payload_match:
        payload_size = int(payload_match.group(1))
        print(f"Payload size: {payload_size} bytes")
    else:
        print("Payload size not found in the trace file. Please check the file contents.")

Payload size: 30 bytes
```

Figure 39. Payload size comparison between employees and departments for a POST request on the Express framework.



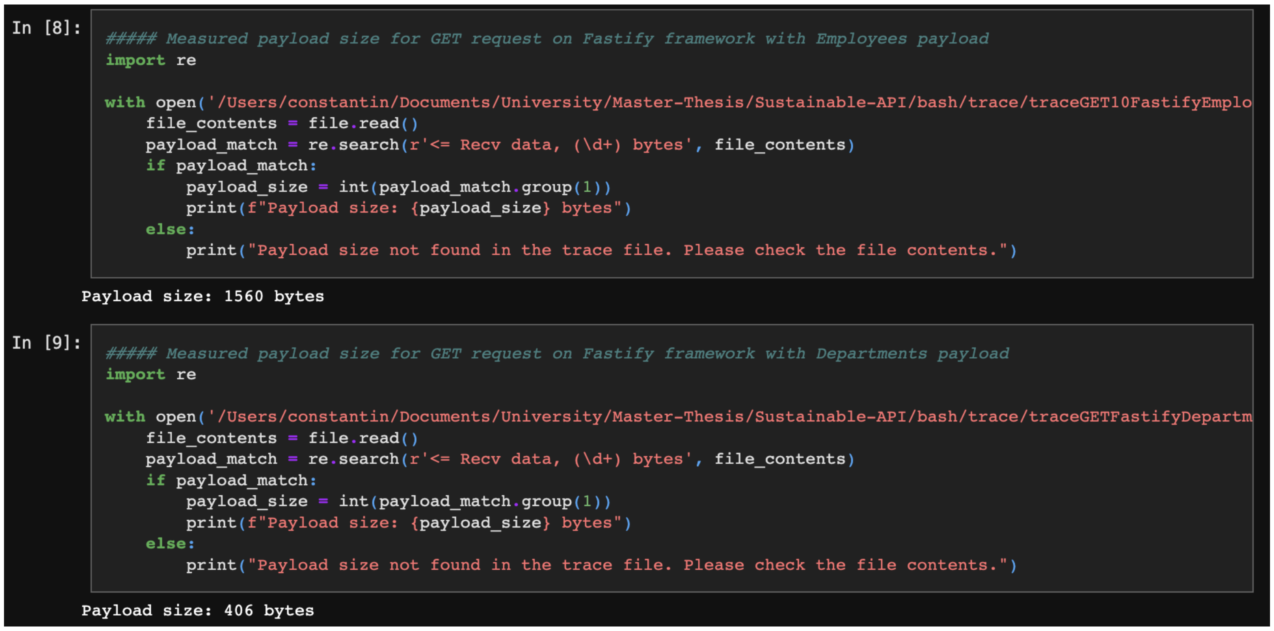
```
In [3]:  ##### Measured payload size for GET request on Express framework with Employees payload
         import re

         with open('/Users/constantin/Documents/University/Master-Thesis/Sustainable-API/bash/trace/traceGET10ExpressEmplo
             file_contents = file.read()
             payload_match = re.search(r'<= Recv data, (\d+) bytes', file_contents)
             if payload_match:
                 payload_size = int(payload_match.group(1))
                 print(f"Payload size: {payload_size} bytes")
             else:
                 print("Payload size not found in the trace file. Please check the file contents.")

        Payload size: 1560 bytes

In [13]: ##### Measured payload size for GET request on Express framework with Departments payload
         import re

         with open('/Users/constantin/Documents/University/Master-Thesis/Sustainable-API/bash/trace/traceGETExpressDepartm
             file_contents = file.read()
             payload_match = re.search(r'<= Recv data, (\d+) bytes', file_contents)
             if payload_match:
                 payload_size = int(payload_match.group(1))
                 print(f"Payload size: {payload_size} bytes")
             else:
                 print("Payload size not found in the trace file. Please check the file contents.")

        Payload size: 451 bytes
```

Figure 40. Payload size comparison between employees and departments for a GET (10 rows) request on Express framework.



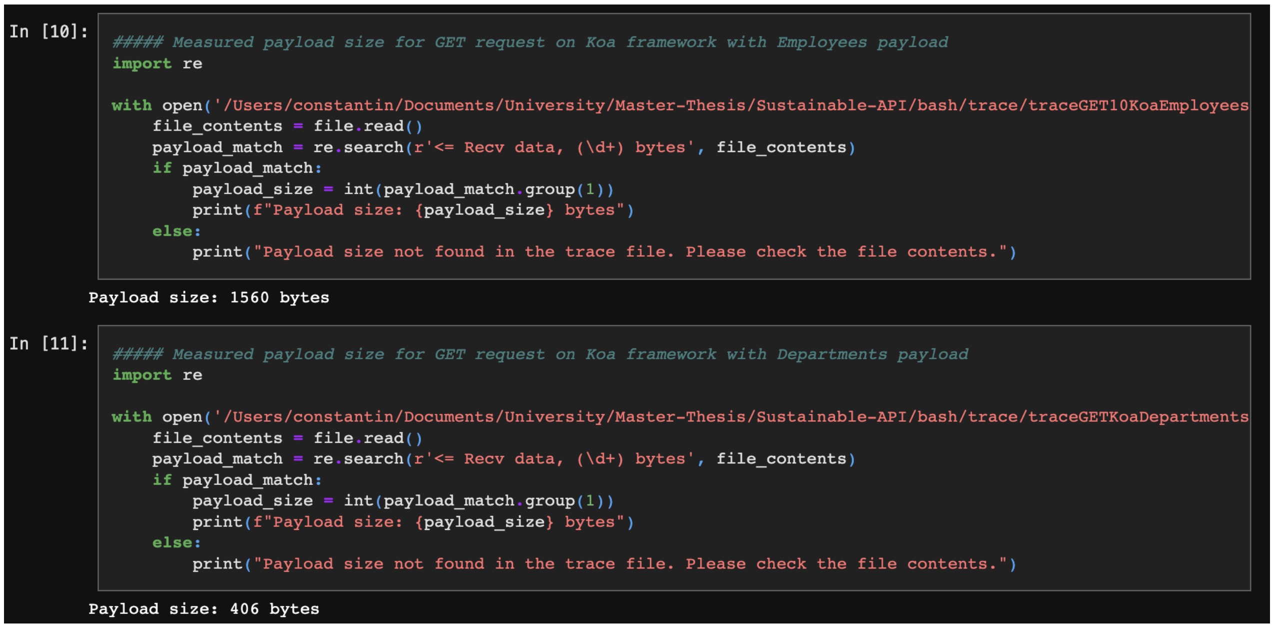
```
In [5]:  ##### Measured payload size for GET request on Nest framework with Employees payload
         import re

         with open('/Users/constantin/Documents/University/Master-Thesis/Sustainable-API/bash/trace/traceGET10NestEmployee
             file_contents = file.read()
             payload_match = re.search(r'<= Recv data, (\d+) bytes', file_contents)
             if payload_match:
                 payload_size = int(payload_match.group(1))
                 print(f"Payload size: {payload_size} bytes")
             else:
                 print("Payload size not found in the trace file. Please check the file contents.")

        Payload size: 1540 bytes

In [6]:  ##### Measured payload size for GET request on Nest framework with Departments payload
         import re

         with open('/Users/constantin/Documents/University/Master-Thesis/Sustainable-API/bash/trace/traceGETNestDepartment
             file_contents = file.read()
             payload_match = re.search(r'<= Recv data, (\d+) bytes', file_contents)
             if payload_match:
                 payload_size = int(payload_match.group(1))
                 print(f"Payload size: {payload_size} bytes")
             else:
                 print("Payload size not found in the trace file. Please check the file contents.")

        Payload size: 406 bytes
```

Figure 41. Payload size comparison between employees and departments for a GET (10 rows) request on the Nest framework.



```
In [8]:  ##### Measured payload size for GET request on Fastify framework with Employees payload
         import re

         with open('/Users/constantin/Documents/University/Master-Thesis/Sustainable-API/bash/trace/traceGET10FastifyEmplo
             file_contents = file.read()
             payload_match = re.search(r'<= Recv data, (\d+) bytes', file_contents)
             if payload_match:
                 payload_size = int(payload_match.group(1))
                 print(f"Payload size: {payload_size} bytes")
             else:
                 print("Payload size not found in the trace file. Please check the file contents.")

         Payload size: 1560 bytes

In [9]:  ##### Measured payload size for GET request on Fastify framework with Departments payload
         import re

         with open('/Users/constantin/Documents/University/Master-Thesis/Sustainable-API/bash/trace/traceGETFastifyDepartm
             file_contents = file.read()
             payload_match = re.search(r'<= Recv data, (\d+) bytes', file_contents)
             if payload_match:
                 payload_size = int(payload_match.group(1))
                 print(f"Payload size: {payload_size} bytes")
             else:
                 print("Payload size not found in the trace file. Please check the file contents.")

         Payload size: 406 bytes
```

Figure 42. Payload size comparison between employees and departments for a GET (10 rows) request on the Fastify framework.



```
In [10]: ##### Measured payload size for GET request on Koa framework with Employees payload
         import re

         with open('/Users/constantin/Documents/University/Master-Thesis/Sustainable-API/bash/trace/traceGET10KoaEmployees
             file_contents = file.read()
             payload_match = re.search(r'<= Recv data, (\d+) bytes', file_contents)
             if payload_match:
                 payload_size = int(payload_match.group(1))
                 print(f"Payload size: {payload_size} bytes")
             else:
                 print("Payload size not found in the trace file. Please check the file contents.")

         Payload size: 1560 bytes

In [11]: ##### Measured payload size for GET request on Koa framework with Departments payload
         import re

         with open('/Users/constantin/Documents/University/Master-Thesis/Sustainable-API/bash/trace/traceGETKoaDepartments
             file_contents = file.read()
             payload_match = re.search(r'<= Recv data, (\d+) bytes', file_contents)
             if payload_match:
                 payload_size = int(payload_match.group(1))
                 print(f"Payload size: {payload_size} bytes")
             else:
                 print("Payload size not found in the trace file. Please check the file contents.")

         Payload size: 406 bytes
```

Figure 43. Payload size comparison between employees and departments for a GET (10 rows) request on the Koa framework.

7.6. Experimental boundaries

This subchapter refers to the limits and constraints that we encountered when designing, conducting, and interpreting our experiment. Each boundary is different as it could be physical, theoretical, or practical, and each plays a critical role in shaping the scope and validity of our scientific research. Knowing these limitations can impact how the findings are interpreted and generalized. Ultimately, navigating experimental boundaries requires careful consideration and critical thinking to ensure that scientific research is conducted and communicated responsibly.

7.6.1. Number of clients

The number of clients a server has to handle at any given time is not only crucial for its performance but also affects its energy consumption.

In our experiment, the server had only one client to deal with, which may have affected the results. In a real-world scenario, multiple clients would be sending requests simultaneously, which would increase the server’s workload and energy consumption by having to prioritize incoming requests, handle multiple data streams, and allocate resources accordingly. The increased load can lead to longer processing times, which requires the server to consume more energy to perform the necessary computations. Additionally, studying the behavior of multiple clients interacting with the same server could shed light on the system's energy management strategies and its ability to allocate resources efficiently. By analyzing the energy consumption metrics of concurrent requests, we could gain a deeper understanding of how the server handles simultaneous interactions and whether it can effectively distribute and utilize energy resources to maintain desired performance levels.

In summary, while our experiments focused on a single client-server interaction, further exploration of concurrent or parallel requests would provide valuable insights into the system's energy consumption under increased load and help assess its energy efficiency and resource management capabilities. This knowledge is crucial for optimizing energy usage and designing more sustainable server-client architectures. To accurately measure the energy consumption of a server in such conditions, it is necessary to simulate the actual operating conditions and workload of a production system, including multiple clients, which leads to a more complex experimental setup and more time for preparation.

7.6.2. Network traffic

The network traffic is another critical factor that affects the energy consumption of a system. In our experiment, the network traffic was isolated with a network cable and not analyzed together with the power consumption, essentially removing this variable from our consideration. However, in a real-world scenario, network traffic is an important factor that needs to be measured and optimized to ensure optimal performance, as it can have a significant impact on the energy consumption of the server and the network infrastructure. When multiple clients are simultaneously sending and receiving data, the network can become congested, leading to increased energy consumption due to higher data transfer rates, more packets being processed, and more network equipment being utilized. Accurately measuring the impact of network traffic on energy consumption might prove crucial for identifying potential bottlenecks and performance issues, and enabling system designers to optimize their solutions to deliver the best possible energy efficiency. Network optimization techniques such as load balancing, packet prioritization, and network compression can help reduce the energy consumption of a system while still maintaining high-performance levels.

7.6.3. Database

The choice of the database can have a significant effect on the experimental results, as different databases have different performance characteristics and limitations. In our experiment, we used the public “Employees” MSSQL database because it was freely available and contained a large amount of data. However, the use of a public database can limit the applicability of our results to real-world scenarios, as the database may not accurately reflect the characteristics of a custom-built database tailored to a specific application or company. Additionally, the use of a free database may also introduce limitations (and indeed, so was ours in terms of storage capabilities, transaction possibilities, and database credits), which can affect the accuracy and reliability of our experimental results. Consequently, we acknowledge that the choice of the database was dictated by practical considerations such as cost, availability, and compatibility with existing systems, thus creating another boundary for our experiment technologies.

The experimental results are significantly influenced by the selection of entities or payloads from the “Employees” database. In our study, we made a deliberate decision to choose an entity with a larger payload and another entity with a smaller payload. For the larger payload, we selected the employees entity because it had the highest number of fields (6) among the available entities in the database. Conversely, for the smaller payload, we chose the departments entity, which had the lowest number of fields (2) and the fewest number of rows. However, it is important to acknowledge that our choice of entities was based on our thoughts and expectations based on the difference in the number of attributes and rows. We should also mention that there were also tables with a larger number of rows but with fewer fields, such as the salaries table.

7.6.4. Amount of data

The amount of data collected in an experiment is critical for the accuracy and reliability of the results obtained. Incomplete or biased data can lead to incorrect conclusions or unreliable performance estimates, which can have severe consequences in real-world applications. In our experiment, we only managed to collect data that fit within our timeframe and according to the methodology described in the “4.3.4. Automated script” subchapter, which may not have been sufficient to fully capture the behavior and performance of the system. For example, the experiment may not have captured rare but important events that could significantly affect the system's performance or behavior. Similarly, the use of an automated script to collect the data can introduce limitations, such as the inability to capture certain aspects of the system's behavior or the introduction of bias into the data collection process. To address these issues, one needs to carefully consider the amount and quality of the data collected in an experiment. The selection of input parameters, the choice of workload, and the exclusion of outliers can all have a significant impact on the results obtained. Additionally, efforts should be made to increase the amount and diversity of the data collected, such as by using different input parameters or varying the workload. However, collecting additional data can be costly and time-consuming, and may not always be feasible within the constraints of the experiment. Therefore, we needed to strike a balance between the amount of data collected and the resources available, while ensuring that the data collected is still representative, reliable, and useful for real-world scenarios.

7.6.5. Hardware setup

The hardware setup that we had for this experiment comes out as another limitation. As mentioned in chapter “4.2. Hardware”, to measure the energy consumption we used a Siglent SPD3303X-3 programmable power supply, and 2 Raspberry Pi 4 Model B, one for the client and one for the server. As for memory storage, the client had a Sandisk 16GB MicroSD card while the server had a Sandisk 64GB MicroSD card. The results are constrained by the hardware that we have used for the experiments, which is the source of the limitation. Different devices or hardware capabilities might have offered different outcomes. As a result, the server's performance can be influenced by the device that hosts it, and if processing a request, for example, takes longer than expected, this can also have an impact on energy usage.

7.6.6. Software setup

The software setup also had an impact on our research. We will start with the Java program called TestController, which we used to measure and gather data on energy consumption using the previously addressed programmable power supply. We are unable to confirm the complete accuracy of the data when using this logging application. Although we believe the data to be roughly accurate, we discovered that when TestController.java was configured to request measurements more frequently than once per second, the measurement frequency decreased over time, and sporadically, TestController.java was unable to keep up with the programmable power supply, resulting in the skipping of some measurements. These factors could potentially impact the accuracy and reliability of the results obtained from the experiment. To address these issues, further investigation is necessary to better understand the communication process between TestController.java and the programmable power supply to ensure that the data collected is comprehensive and accurate.

Another limitation is given by the way we have chosen the 4 Node.js frameworks that were used to compare their energy efficiency. As described in chapter “4.1. Node.js frameworks”, we have picked the frameworks based on their GitHub stars popularity. Initially, the top 4 frameworks we picked for our comparison were Express.js, Nest, Strapi, and Meteor. However, we soon realized that we could not use Meteor as the framework is not compatible with the Raspberry Pi due to the ARM architecture of the processor. This incompatibility made us choose the next most popular framework which was Koa.js. Similarly, after we established the database connection on Microsoft Azure as described in the subchapter “4.3.2. Database”, we realized that Strapi was not compatible with MSSQL, which again, forced us to choose the next popular framework (i.e. the 6th from the list), Fastify.

7.6.7. Raspberry Pi idle consumption

The Raspberry PI's idle power usage places further limitations on our research. The server and client were each running in headless mode, where the server was supposed to run the RESTful

APIs, handle the requests, and send back the response, while the client was supposed to only make requests to the server, by running a script. We had to make an effort to exclude any other background processes to obtain valid results with little to no effect on energy consumption. As a result, we shut down all of the Raspberry Pi's applications as explained in the subchapter “4.3.4. Automated script”, leaving only the terminal open, making sure that Bluetooth, Wi-Fi, and automatic updates were turned off, using an ethernet connection to the network switch, and doing so without the use of monitors. However, we do not know if and what other critical or non-critical processes for the operating system were running in the background, without us being able to turn them off. Thus, we suspect that our results might have been influenced by certain hidden background processes.

7.6.8. Human errors and bias

In addition to the aforementioned factors, there are several other sources of human errors and biases that might have affected the results of our experiment. One such factor is the presence of experimenter bias. Our own preconceived notions, expectations, or beliefs about the Node.js frameworks being compared could have subtly influenced our approach, data interpretation, and conclusions. This bias might have inadvertently influenced the way we collected and analyzed the data, potentially skewing the results.

Moreover, the experimental setup itself may have introduced certain limitations or biases. For instance, the specific configuration of the software and hardware, network environment, or testing conditions could have inadvertently favored or disadvantaged certain frameworks. Unintended variations in these factors might have influenced the observed energy consumption patterns, leading to potential biases in the results.

Furthermore, the process of data collection and analysis involves numerous subjective decisions and judgment calls. Choices such as data filtering, outlier handling, or statistical methods employed can introduce unintentional biases. Even the rounding or approximation of numerical values during calculations can impact the final results. It is crucial to be aware of these potential biases and take steps to minimize their impact by applying rigorous and standardized procedures.

Additionally, human errors in the form of data preparation or recording mistakes can occur, leading to inaccuracies in the collected data. These errors might go unnoticed during the data analysis phase and could potentially introduce noise or distortion into the results.

Awareness of these limitations and potential sources of errors allows us to interpret the results with a critical eye. It emphasizes the importance of replicating the experiment under different conditions and conducting further studies to validate the findings, as it is impossible to eliminate all human errors and biases from our experimental process. By striving for transparency, rigor, and continual improvement in our experimental approaches, we can enhance the reliability and robustness of our scientific investigation.

7.7. Final remarks

This study has provided a comprehensive comparison of the energy consumption of popular Node.js frameworks, including Express, Nest, Fastify, and Koa when utilized for developing RESTful APIs. The findings shed light on the energy efficiency aspects of these frameworks and may offer valuable insights for developers, organizations, and the broader software development community.

Throughout the analysis, it became evident that there are noticeable variations in energy consumption among the different frameworks. These variations can be attributed to various factors such as architectural design, code optimizations, internal mechanisms, and resource utilization.

The results indicate that frameworks like Nest tend to exhibit higher energy consumption compared to Fastify, Koa, or Express. This finding suggests that developers seeking energy-efficient solutions may consider leveraging one of the latter 3 frameworks for their RESTful API projects. However, it is crucial to note that the choice of the framework should not solely rely on energy consumption but should also consider other factors such as performance, scalability, and built-in features.

It is worth emphasizing that the observed energy consumption differences are specific to the context of RESTful API development and may vary in other application scenarios. Therefore, it is recommended to conduct further research and evaluations in different use cases and real-world scenarios to validate the findings and draw more comprehensive conclusions.

Moving forward, there are several paths for future exploration in this area. First and foremost, conducting more extensive experiments with larger datasets and diverse workloads would enhance the robustness and reliability of the results. Additionally, investigating the impact of different configuration settings (changes to the number of iterations), deployment environments (using a cloud provider), and third-party dependencies (e.g. databases, network, devices used to test, hardware infrastructure) on energy consumption would provide a more nuanced understanding of the factors influencing energy efficiency. These factors are known to have an influence on the energy consumption of cloud applications, as seen in (Li et al., 2017).

Moreover, integrating energy optimization strategies and techniques into the development process of Node.js frameworks could potentially lead to significant improvements in energy efficiency. This includes exploring methods for optimizing code execution, reducing unnecessary resource consumption, and adopting energy-aware architectural patterns.

By fostering awareness and promoting energy-conscious development practices within the Node.js community, we can collectively contribute to a more sustainable and environmentally friendly web ecosystem. Encouraging collaboration, knowledge sharing, and open dialogue among developers, researchers, and industry stakeholders will be crucial in driving these initiatives forward.

In conclusion, this study highlights the importance of considering energy consumption as a key factor in the selection of Node.js frameworks for the development of RESTful APIs. By making informed decisions and embracing energy-efficient practices, we have the opportunity to reduce the environmental impact of web applications while maintaining performance and delivering exceptional user experiences.

8. Conclusion

In this thesis, our primary objective was to investigate the energy efficiency of the top most popular Node.js frameworks. To accomplish this, we conducted research that involved a comparison of the frameworks being used to develop RESTful APIs.

In order to answer the first research question of our study, we utilized the context of RESTful APIs as a practical use case. We aimed to investigate the feasibility of evaluating energy consumption by examining the measurement possibilities of the different frameworks within this context. Additionally, we examined the practices and techniques necessary to ensure a fair and valid comparison in the domain of Green Software. These included multiple iterations with breaks to prevent hardware overheating and ensure clearer differentiation between requests. Additionally, we minimized background processes and avoided any additional computations for displays or monitors during the software execution. To measure energy consumption, we employed a setup consisting of a programmable power supply integrated with data logging software, utilizing 2 Raspberry Pis for running the server and client. The orchestration of requests was automated using a script, while all devices were connected via cable to the same network.

Regarding the answer to the second research question, the experimental findings revealed significant differences among the examined Node.js frameworks. Koa demonstrated superior efficiency when handling larger payloads, while Fastify exhibited higher efficiency with smaller payloads, with Express ranking third overall. On the other hand, Nest consistently displayed the worst efficiency across all scenarios. Notably, the server component consumed more energy than the client, as it processed requests and delivered responses. Surprisingly, the smaller payload occasionally consumed more than double the time and energy compared to the larger payload.

Through the adoption of various methodologies and the undertaking of comprehensive experiments, our contributions to the realm of Green Software have yielded significant results. Specifically, we looked into the energy consumption characteristics of diverse Node.js frameworks, acquiring valuable insights that contribute to the advancement of sustainable software development practices. Our efforts extend beyond mere analysis, as we strive to bring awareness among software developers regarding the broader implications of their creations. By emphasizing energy consumption as a crucial factor in improving ICT carbon footprints, we aim to expand the field of Green IT, transcending the focus on underlying application construction. By integrating these principles into the consciousness of developers and stakeholders alike, we aspire to drive positive change and pave the way for a more environmentally conscious approach to software development.

8.1. Future research

This thesis offers numerous possibilities for future research based on its findings. These opportunities encompass both conducting follow-up experiments to expand upon the current results and conducting new experiments to enhance the existing knowledge base. Furthermore, the thesis has also brought to light certain limitations in the conducted experiments, underscoring the need for further investigation. In this section, we will present a selection of potential projects that can serve as a continuation of this thesis, aside from the ones already mentioned in ”7.6. Experimental boundaries” and “7.7. Final remarks”.

8.1.1. Green labels

In the realm of future research, one promising avenue is the investigation of green labels for Node.js frameworks, perhaps utilizing similar criteria as in the EcoSoft framework proposed by Deneckère and Rubio in (Deneckère & Rubio, 2020). The development of standardized labels specifically tailored to Node.js frameworks could provide valuable information about their energy efficiency, aiding developers and software consumers in making informed choices. By adapting the criteria from EcoSoft, which consider factors such as runtime energy efficiency, future data conversion, or the sustainability of documentation and specification, it would be possible to create an assessment system for Node.js frameworks.

8.1.2. Other Node.js frameworks used for developing APIs

For this research, due to time constraints, we were able to compare the energy consumption of only 4 Node.js frameworks. As mentioned earlier in chapter “4.1. Node.js frameworks”, these frameworks were selected based on their popularity measured by the number of stars on GitHub. However, the scope of this study could be extended to include a broader range of Node.js frameworks, such as Meteor or Strapi, which were not included in our analysis due to the limitations mentioned in chapter “7.6.6. Software setup”.

Expanding the comparison to include additional Node.js frameworks would provide a more comprehensive understanding of the energy efficiency across a wider spectrum of options available to developers. Meteor, for instance, is a full-stack framework that offers real-time updates and a wide range of features, while Strapi is known for its flexibility and extensibility as a headless CMS (Content Management System).

8.1.3. Extending the usage of RESTful APIs

By utilizing RESTful APIs in a comprehensive manner, such as incorporating them into a complete application that encompasses both backend and frontend aspects, it becomes possible to evaluate the associated energy consumption. This evaluation can be achieved by allowing users to perform actions on the user interface (UI) that trigger corresponding actions on the server. In our case, we employed a bash script to automate the client's interaction with the server. Expanding the scope of this usage, browser automation tools like Selenium[29] can be employed to automate the interaction with the frontend.

[29] Selenium’s website: https://www.selenium.dev/

9. References

1. Adhikari, A., DeNero, J., & Wagner, D. (2019). Computational and inferential thinking: The foundations of data science. *University of California, Berkeley.*

2. Adriano Zanin Zambom, & Dias, R. (2013). A Review of Kernel Density Estimation with Applications to Econometrics. *International Econometric Review*, *5*(1), 20–42.

3. Andrae, A. S. G. (2020). Hypotheses for Primary Energy Use, Electricity Use and CO2 Emissions of Global Computing and Its Shares of the Total Between 2020 and 2030. *WSEAS TRANSACTIONS on POWER SYSTEMS*, *15*(4), 50–59. https://doi.org/10.37394/232016.2020.15.6

4. Bangare, S., Gupta, S., Dalal, M., & Inamdar, A. (2016). Using Node.Js to Build High Speed and Scalable Backend Database Server. *International Journal of Research in Advent Technology*, *4*, 19.

5. Beck, K. (1999). Embracing Change with Extreme Programming. *Computer*, *32*(10), 70–77. https://doi.org/10.1109/2.796139

6. Becker, N., Venters, C., Becker, C., Penzenstadler, B., Chitchyan, R., Seyff, N., Duboc, L., & Easterbrook, S. (2015). Sustainability design and software: The karlskrona manifesto. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (Vol. 2, pp. 467-476). IEEE.

7. Belkhir, L., & Elmeligi, A. (2018). Assessing ICT global emissions footprint: Trends to 2040 & recommendations. *Journal of Cleaner Production*, *177*(177), 448–463. https://doi.org/10.1016/j.jclepro.2017.12.239

8. Bennett, K., & Rajlich, V. (2000). Software Maintenance and Evolution: A Roadmap. *Proceedings of the Conference on the Future of Software Engineering*, 73–87.

9. Borges, H., & Valente, M. T. (2018). What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software 146 (2018): 112-129*.

10. Brown, M. (2001). Power supply cookbook. ser. *EDN Series for Design Engineers. Newnes, 212*.

11. Calero, C., Mancebo, J., Garcia, F., Moraga, M. A., Berna, J. A. G., Fernandez-Aleman, J. L., & Toval, A. (2020). 5Ws of green and sustainable software. *Tsinghua Science and Technology*, *25*(3), 401–414. https://doi.org/10.26599/tst.2019.9010006

12. Cruz, L., & Abreu, R. (2021). On the Energy Footprint of Mobile Testing Frameworks. *IEEE Transactions on Software Engineering*, *47*(10), 2260–2271. https://doi.org/10.1109/tse.2019.2946163

13. Demashov, D., & Gosudarev, I. (2019). Efficiency Evaluation of Node.js Web-Server Frameworks. In *MICSECS*. 2019.

14. Deneckère, R., & Rubio, G. (2020). EcoSoft: Proposition of an Eco-Label for Software Sustainability. In *Advanced Information Systems Engineering Workshops: CAiSE 2020 International Workshops, Grenoble, France, June 8–12, 2020, Proceedings 32 (pp. 121-132).* Springer International Publishing. https://doi.org/10.1007/978-3-030-49165-9_11

15. Eben Upton, & Gareth Halfacree. (2016). Raspberry Pi user guide. John Wiley & Sons.

16. Eder, K., Gallagher, J., López-García, P., Muller, H. L., Zorana Bankovic, Georgiou, K., Rémy Haemmerlé, Hermenegildo, M. V., Bishoksan Kafle, Kerrison, S., Maja Hanne Kirkeby, Maximiliano Klemen, Li, X., Umer Liqat, Morse, J., Morten Rhiger, & Rosendahl, M. (2016). ENTRA: Whole-systems energy transparency. *Microprocessors and Microsystems*, *47*, 278–286. https://doi.org/10.1016/j.micpro.2016.07.003

17. Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. *University of California*, *Irvine*.

18. Flanagan, D., & Novak, G. M. (1998). Java-Script: The Definitive Guide, Second Edition. *Computers in Physics*, *12*(1), 41. https://doi.org/10.1063/1.168647

19. Harrington, J. L. (2016). Relational Database Design and Implementation (4th ed.). *Morgan Kaufmann*.

20. Jennifer Niederst Robbins. (2018). Learning web design : a beginner's guide to HTML, CSS, Javascript, and web graphics. *O'Reilly Media, In*.

21. Kern, E., Hilty, L. M., Guldner, A., Maksimov, Y. V., Filler, A., Gröger, J., & Naumann, S. (2018). Sustainable software products—Towards assessment criteria for resource and energy efficiency. *Future Generation Computer Systems*, *86*, 199–210. https://doi.org/10.1016/j.future.2018.02.044

22. Kirkeby, M., Krabben, T., Larsen, M., Mikkelsen, M., Rosendahl, M., Sundman, M., Petersen, T., & Schoeberl, M. (2022). Energy Consumption and Performance of Heapsort in Hardware and Software. *arXiv preprint arXiv:2204.03401*.

23. Lane, D. (2016). Introductory Statistics Online Statistics.

24. Li, Z., Selome Kostentinos Tesfatsion, Bastani, S., Ali-Eldin, A., Elmroth, E., Kihl, M., & Ranjan, R. (2017). A Survey on Modeling Energy Consumption of Cloud Applications: Deconstruction, State of the Art, and Trade-Off Debates. *IEEE Transactions on Sustainable Computing*, *2*(3), 255–274. https://doi.org/10.1109/tsusc.2017.2722822

25. Ling, S. J., Sanny, J., Moebs, W., Openstax College, & Rice University. (2016). *University physics* (Vol. 1). Openstax, Rice University.

26. Masse, M. (2011). REST API design rulebook: designing consistent RESTful web service interfaces. *O'Reilly Media, Inc*.

27. Mckinney, W. (2013). Python for data analysis. *O'Reilly Media, Inc*.

28. Naumann, S., Dick, M., Kern, E., & Johann, T. (2011). The GREENSOFT Model: A reference model for green and sustainable software and its engineering. *Sustainable Computing: Informatics and Systems*, *1*(4), 294–304. https://doi.org/10.1016/j.suscom.2011.06.004

29. Patrou, M., B. Kent, K., Siu, J., & Dawson, M. (2021). Energy and Runtime Performance Optimization of Node.js Web Requests. *2021 IEEE International Conference on Cloud Engineering (IC2E)* , 71–82.

30. Penzenstadler, B. (2015). From Requirements Engineering to Green Requirements Engineering. *Green in Software Engineering*, 157–186. https://doi.org/10.1007/978-3-319-08581-4_7

31. Pereira, R., Carcao, T., Couto, M., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). Helping Programmers Improve the Energy Efficiency of Source Code. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. https://doi.org/10.1109/icse-c.2017.80

32. Perkel, J. M. (2018). Why Jupyter is data scientists' computational notebook of choice. *Nature*, *563*(7729), 145–146. https://doi.org/10.1038/d41586-018-07196-1

33. Persson, M. (2020). JavaScript DOM Manipulation Performance: Comparing Vanilla JavaScript and Leading JavaScript Front-end Frameworks. *Faculty of Computing, Blekinge Institute of Technology*.

34. Rani Das, K. (2016). A Brief Review of Tests for Normality. *American Journal of Theoretical and Applied Statistics*, *5*(1), 5. https://doi.org/10.11648/j.ajtas.20160501.12

35. Samuels, M. L., Witmer, J. A., & Schaffner, A. A. (2016). Statistics for the life sciences. *Pearson Education Limited, Cop*.

36. Satheesh, M., D'mello, B. J., & Krol, J. (2015). Web development with MongoDB and NodeJs. *Packt Publishing Ltd*.

37. Singh, R. (1996). International Standard ISO/IEC 12207 Software Life Cycle Processes. *Software Process: Improvement and Practice*, *2*(1), 35–50. https://doi.org/10.1002/(sici)1099-1670(199603)2:1%3C35::aid-spip29%3E3.0.co;2-3

38. Uakarn, C., Chaokromthong, K., & Sintao, N. (2021). Sample Size Estimation using Yamane and Cochran and Krejcie and Morgan and Green Formulas and Cohen Statistical Power Analysis by G*Power and Comparisons. *Apheit International Journal, 10(2), 76-86.*

39. Wolfram, N., Lago, P., & Osborne, F. (2017). Sustainability in software engineering. 2017 Sustainable Internet and ICT for Sustainability (SustainIT). https://doi.org/10.23919/sustainit.2017.8379798

10. Appendix

10.1. Logbook

Date / Period (dd/mm/yyyy)	Goal(s)	Description
1/11/2022	What thesis idea to choose: comparing different libraries, API frameworks, frontend frameworks, design patterns, and improvements in code. Something to include software development and energy consumption comparisons.	We chose to compare API frameworks as benchmarks of different frontend frameworks already exist, we did not find design patterns appealing enough and there are no popular studies comparing APIs and their efficiency. We settled on JavaScript (Node.js) due to its popularity in the industry and our previous experience.
2/2/2023	Decide what to work on for the next 2-3 weeks. Choose what frameworks to code and compare. Find and read relevant literature.	We decided to write for simple REST APIs in the 4 most popular Node.js frameworks according to GitHub stars/ratings (express, nest, strapi, meteor).
2/2/2023	Developing and having working APIs for express and nest. Pair programming.	We started with express and nest as they are the most popular frameworks for Node.js.
7/2/2023	Developing the API in strapi. Looking for benchmark methodologies (tools) and benchmark data for API energy measurements.	We wanted to see if there are any general benchmark guidelines from the literature that apply to measuring the energy consumption APIs.
9/2/2023	Developing the API in meteor. Testing all other APIs on both of our machines.	Finalizing all APIs and making sure all of them work. This would allow us to have a basis for initial measurements before we define the parameters of the experiment.
16/2/2023	Look through the article suggested by Kerstin: James Pallister, Sigenergy, Green Software Foundation.	The goal was to target our research towards (re)sources that might publish about APIs and performance benchmarks.

23/2/2023	Split and write general paragraphs for the thesis. Find and choose benchmark tools and the framework for running the experiment (look at Patrou et. al.)	Decided on the paragraphs to write from the thesis that are not custom to our experiment but rather form the basis (i.e. introduction, background, reasoning). Also looked at different tools on the market for automating the measurements.
02/03/2023	Raspberry PI setup at RUC for both the client and the server	Card reader and different setups on client and server on Raspberry Pis. Cloning the repository and setting it up on the server and making requests from the client, while being on a private network. Setting up the OS on the RP: https://www.tomshardware.com/reviews/raspberry-pi-headless-setup-how-to,6028.html
16/3/2023 - 23/3/2023	Review of the chosen frameworks. Finalize all the APIs for all 4 frameworks.	One of the initial frameworks (meteor) was removed due to the lack of support on ARM architectures (RP) and a second one (strapi) was removed for the lack of MSSQL support. The next 2 most popular frameworks were picked instead (koa and fastify).
30/3/2023	Design the bash setup and run the measurements.	We have separated all requests into 8 files, due to 4 frameworks (express, nest, fastify, koa) and 2 entities (employees which is the bigger payload with 6 fields, and departments with only 2 fields) for each file. We start with an initial baseline measurement by waiting 5 minutes, to ensure no activity is running in the background. For each of the 8 files (i.e. 8 experiments), we execute each file (having 5 requests for GET, GET by

		ID, POST, PUT, DELETE) 30 times (iterations). In between the 30 iterations, we have 1-minute breaks, and then in between the experiments (again, 3 in total for each file), we have a 5-minute baseline break.
6/4/2023 - 13/4/2023	Split the bash script runs for each file.	As the free plan of the DB could not handle well all connections at once, we decided to run each server separately and have the bash script execute each file (between employees and departments) for each framework
14/4/2023 - 30/3/2023	Change the way the script is executed (no break in between but 30 consecutive calls instead). Write Experiment Design in the report	We noticed that the results were not as clear and robust when we had a break after each API call due to the data being closer to the idle consumption, rather than showing and actually logging all the spikes. Because of this, we decided to aggregate all requests in a “unit of work” and after all consecutive requests, we waited 1 minute.
1/5/2023 - 7/5/2023	Split the CSV files.	We have tried to code a script that would automatically split the waiting times from our measurements but due to the same values being equal, part of the measurement data was labeled as being part of the “idle” consumption. Because of this issue, we decided to manually split the CSV files.
8/5/2023 - 15/5/2023	Test and validate the data. Display the results as diagrams. Contact Luis Cruz to get an answer related to our method of calculating the area under the curve. Write Data preparation and analysis in the report.	Create all Python Notebooks and run all necessary tests (A/B, sample size, normality tests). Create all the graphs for data preview and save the needed data into separate files that would be later imported in the Notebooks where we compare the frameworks. Because we were not clear in terms

	Write the Background, Statistical analysis subchapter in the report.	of which method to apply for approximating the energy consumption, we contacted Luis Cruz to get his opinion on the topic. (Luís Cruz – All you need to know about Energy Metrics in Software Engineering (luiscruz.github.io)).
16/5/2023 - 23/5/2023	Write the results in the report. Compare payload sizes. Write a discussion in the report.	Because the smaller payload consumes more than the larger payload, we decided to compare the size of the data sent on the network, to see if our results are correlated with the sizes. Unfortunately, we saw that the employees payload was bigger than the departments ones, hence having no explanation for our payload differences. Finalize the chapters about the results and the discussion in our report.
24/5/2023 - 1/6/2023	Write the conclusion and abstract. Review and edit the entire report.	The final week before the hand-in. The goal was to write the conclusion chapter, together with areas for future research that arose from our limitations and experiment design. Final reviews of the report, where we formatted the text and the pictures, and indexed the figures and tables.

10.2. Software versions

Software	**Version number**
dotenv	16.0.3
mssql	9.1.1
mysql	2.18.1
body-parser	1.20.1
express	4.18.2
fastify	4.14.1
@koa/router	12.0.0
koa	2.14.1
koa-body	6.0.1
koa-router	12.0.0
nodemon	2.0.21
@nestjs/common	9.0.0
@nestjs/core	9.0.0
@nestjs/platform-express	9.0.0
@nestjs/typeorm	9.0.1
reflect-metadata	0.1.13
rxjs	7.2.0
typeorm	0.3.12
@nestjs/cli	9.2.0
@types/express	4.17.13
@types/jest	29.2.4
@types/node	18.11.18
@types/supertest	2.0.11
@typescript-eslint/eslint-plugin	5.0.0

@typescript-eslint/parser	5.0.0
eslint	8.0.1
eslint-config-prettier	8.3.0
eslint-plugin-prettier	4.0.0
jest	29.3.1
prettier	2.3.2
source-map-support	0.5.20
supertest	6.1.3
ts-jest	29.0.3
ts-loader	9.2.3
ts-node	10.0.0
tsconfig-paths	4.1.1
typescript	4.9.5
JavaScript	ECMAScript 2022
curl	7.87.0
Python	3.11.1
Anaconda Navigator	2.4.0
Jupyter Notebook	6.5.2
pandas	2.0.1
numpy	1.24.3
scipy	1.10.1
seaborn	0.12.0
matplotlib	3.7.1
TestController	2.03
MacOS	Ventura 13.3.1
Windows	11 Home 22H2 22621.1778
Raspberry Pi OS 32-bit	11(Bullseye), Kernel: Linux 5.10.92-v7l+

10.3. Data Preview

9.4.1. Express Server Employees

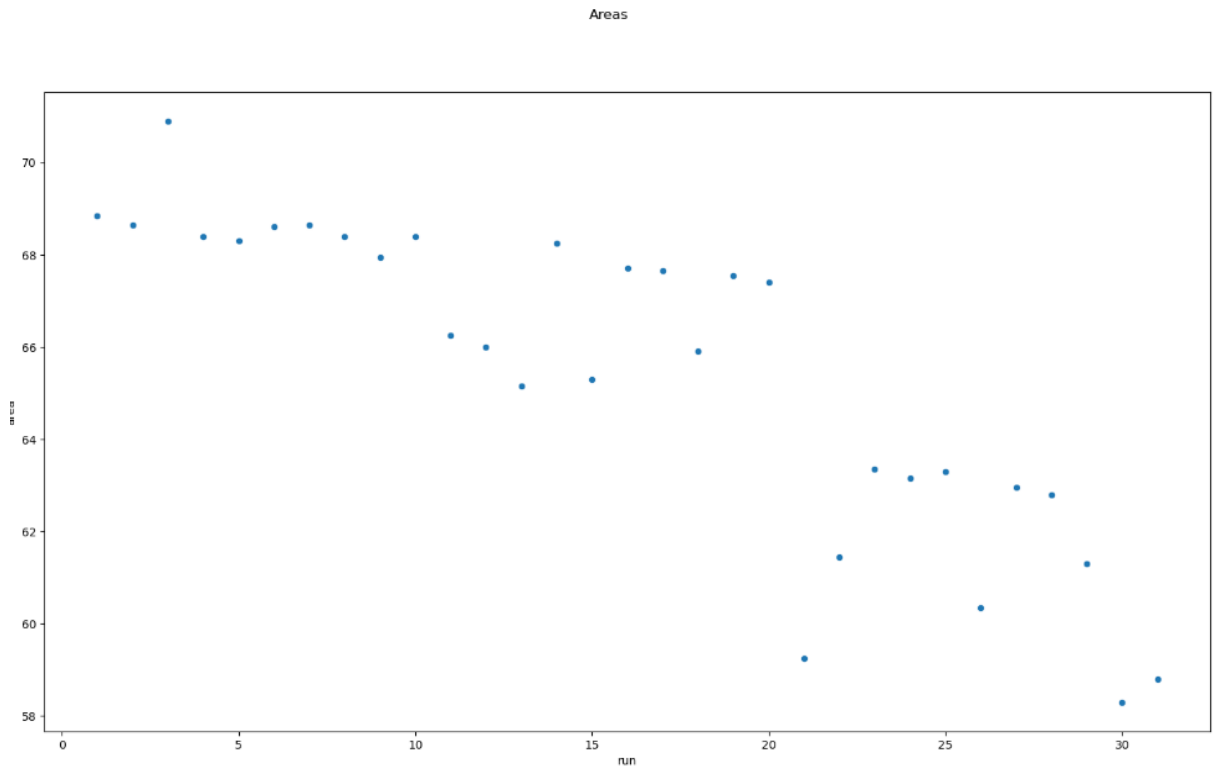


Figure A1. Scatter plot of the areas for Express on the server side using the employees payload.

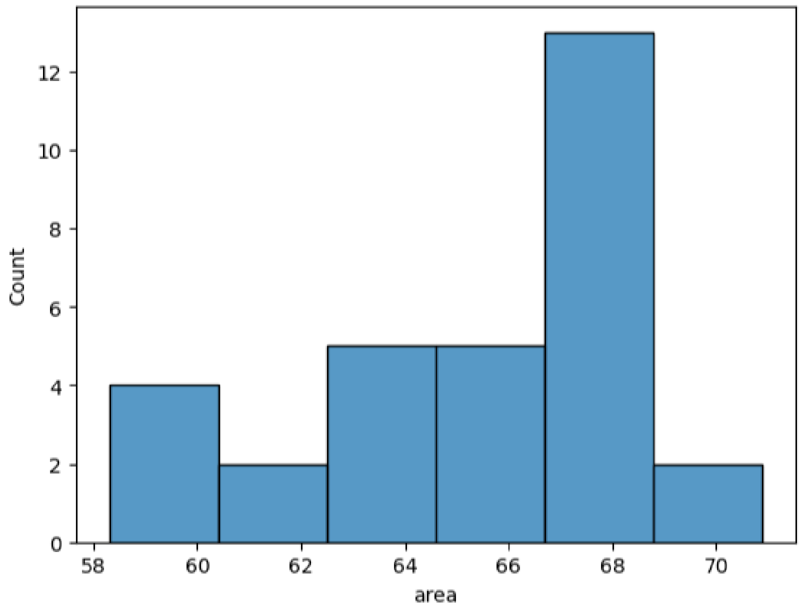


Figure A2. Histogram plot of the areas for Express on the server side using the employees payload.

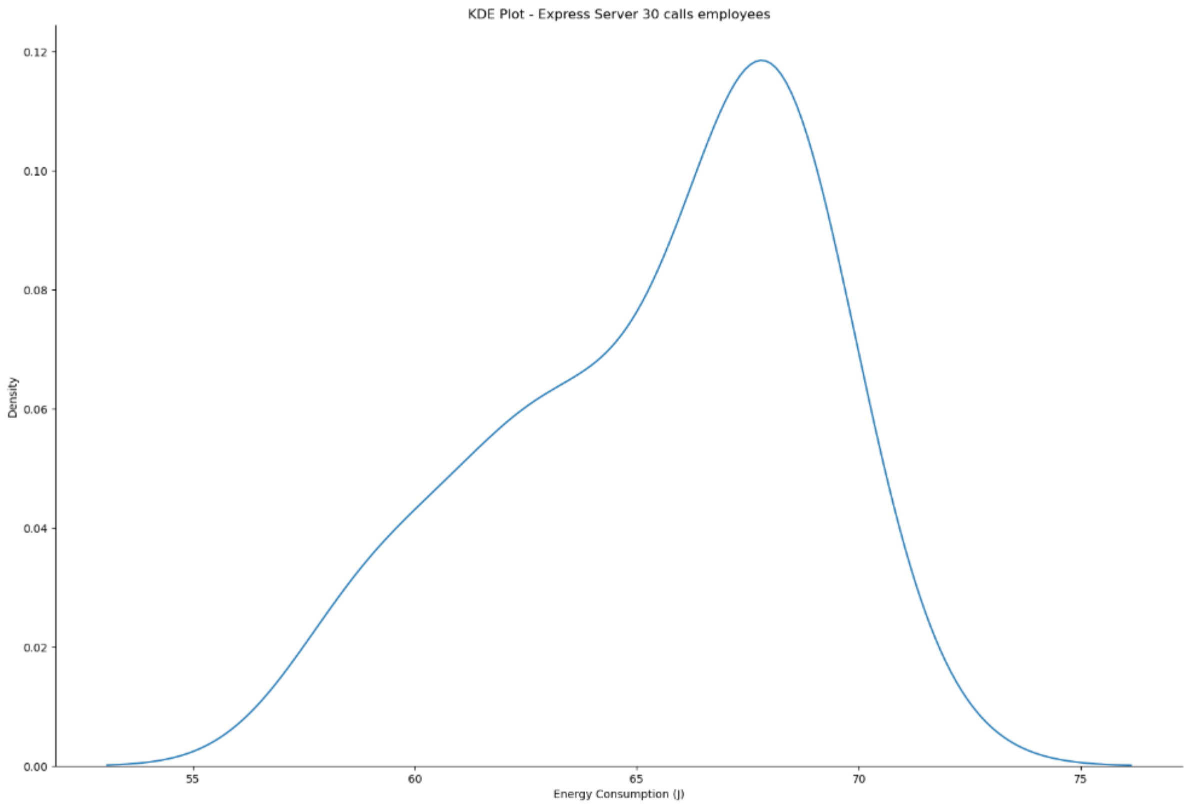


Figure A3. KDE plot of the areas for
Express on the server side using the employees payload.

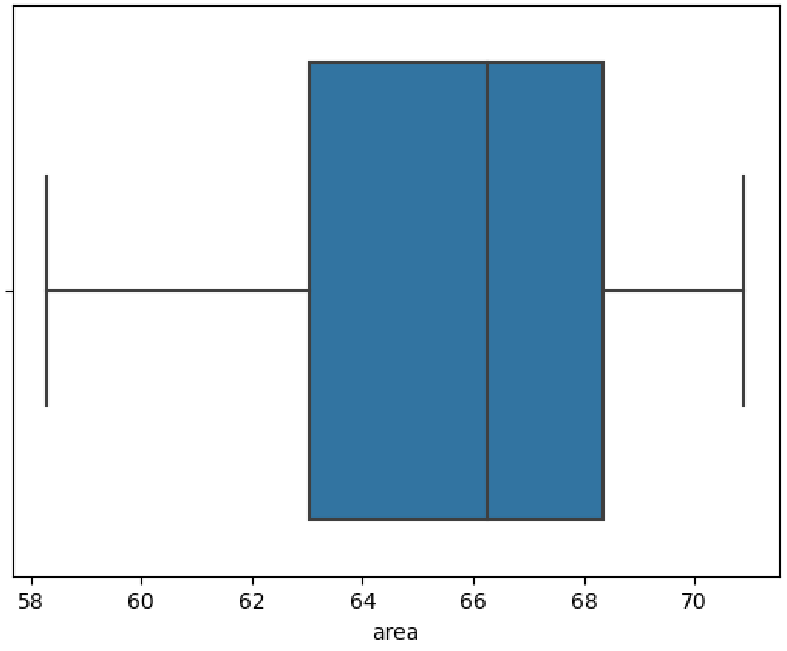


Figure A4. Box plot of the areas for
Express on the server side using the employees payload.

9.4.1. Express Server Departments

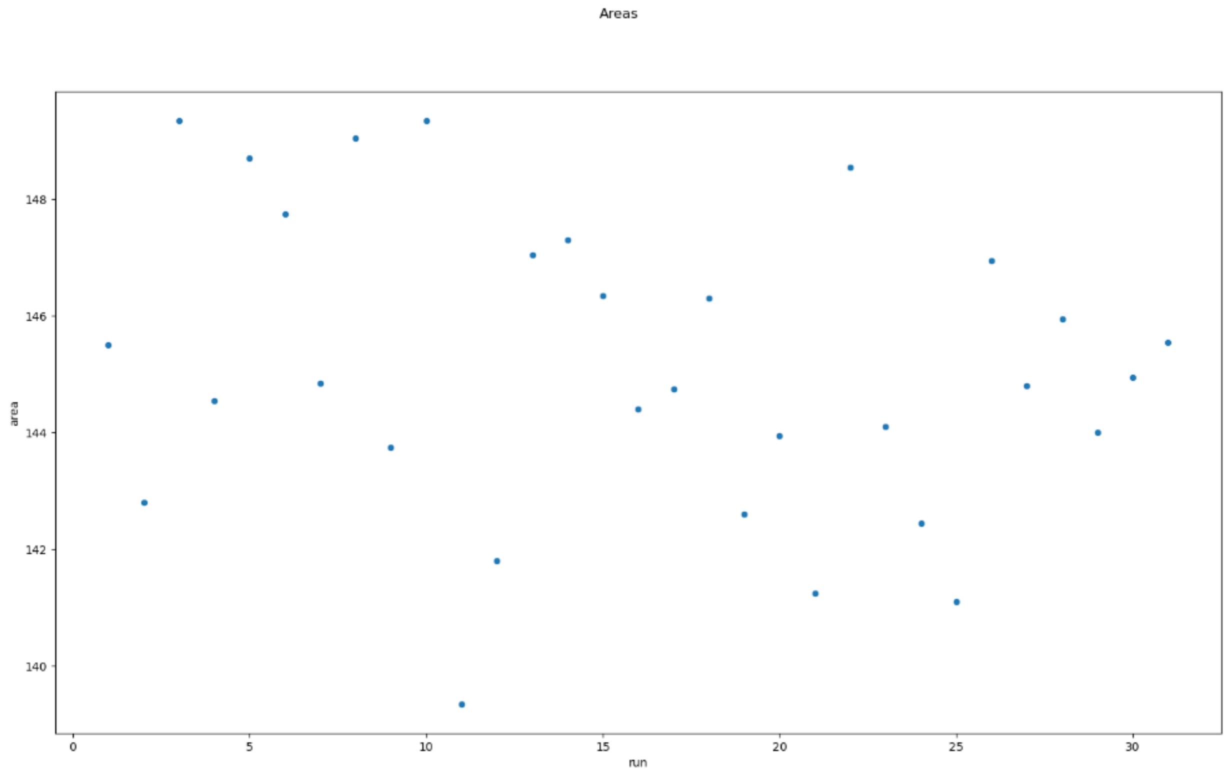


Figure A5. Scatter plot of the areas for Express on the server side using the departments payload.

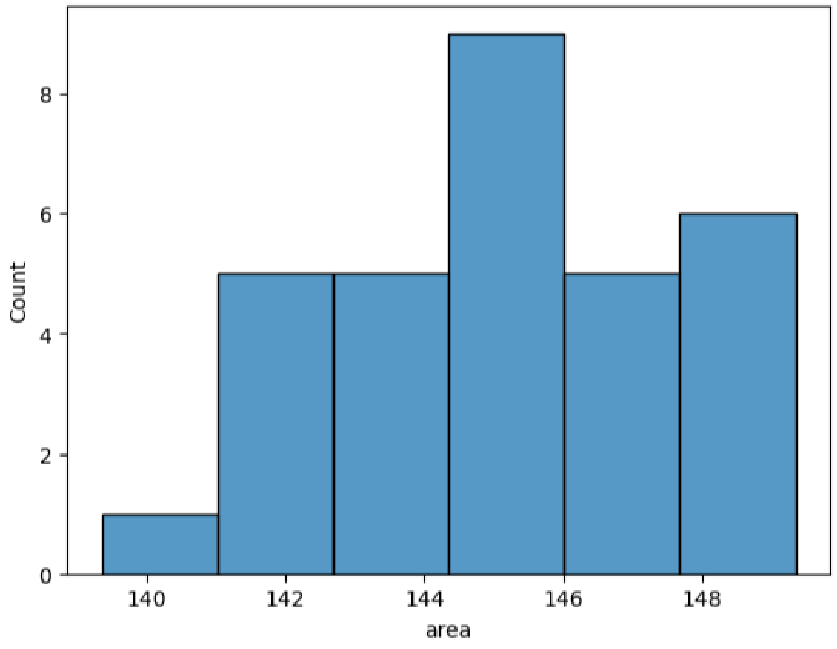


Figure A6. Histogram plot of the areas for Express on the server side using the departments payload.

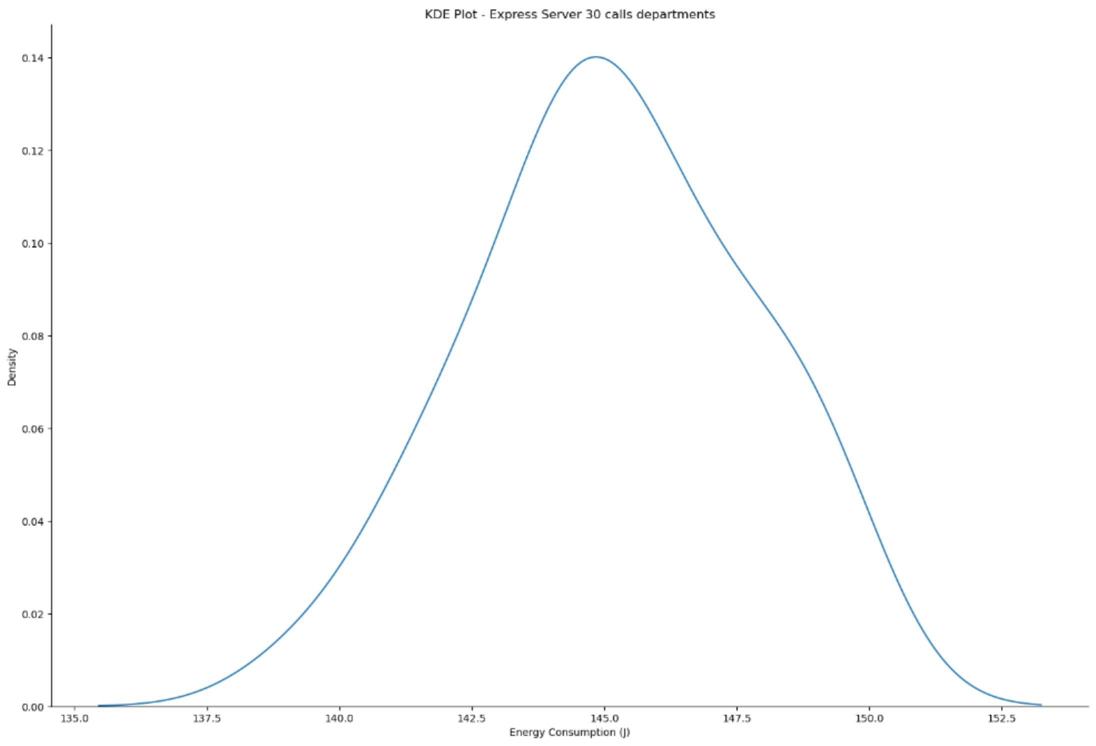


Figure A7. KDE plot of the areas for
Express on the server side using the departments payload.

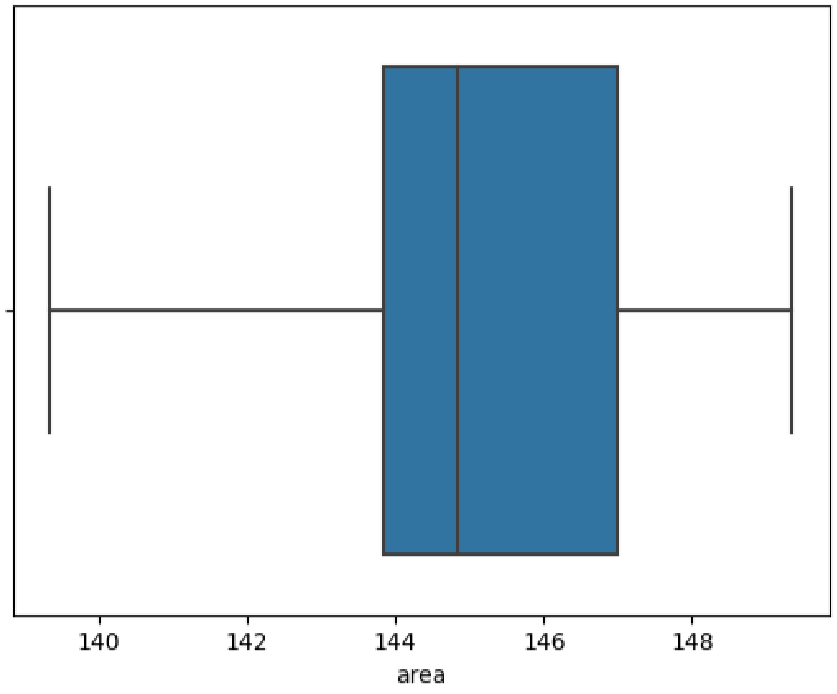


Figure A8. Box plot of the areas for
Express on the server side using the departments payload.

9.4.2. Nest Server Employees

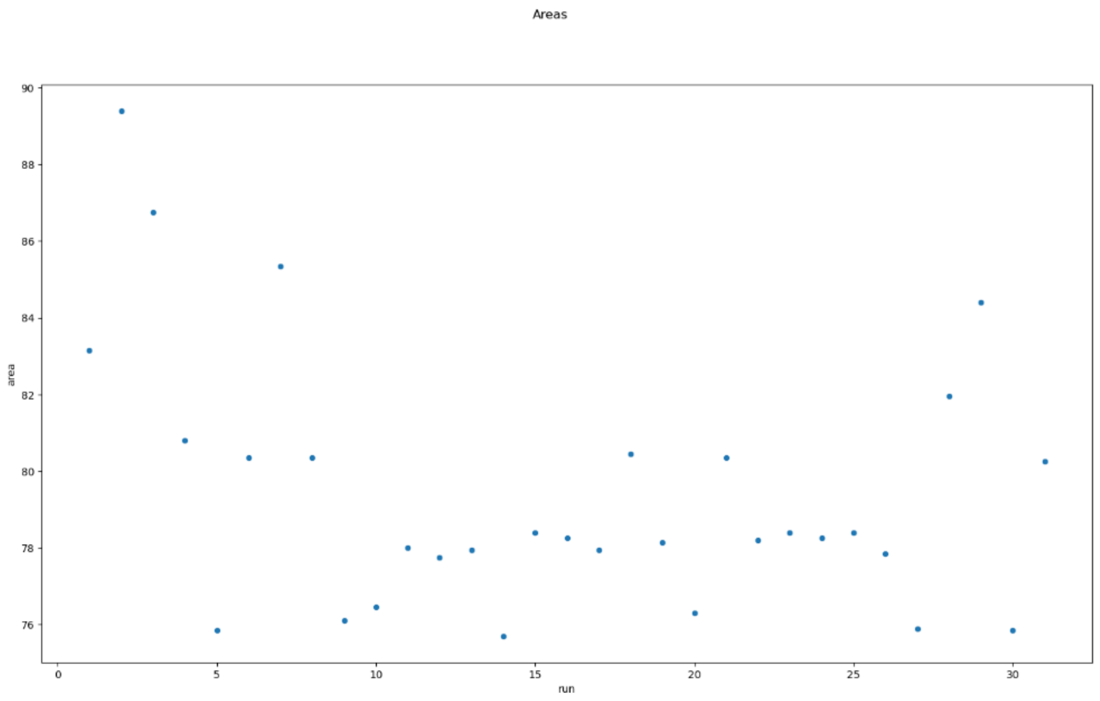


Figure A9. Scatter plot of the areas for Nest on the server side using the employees payload.

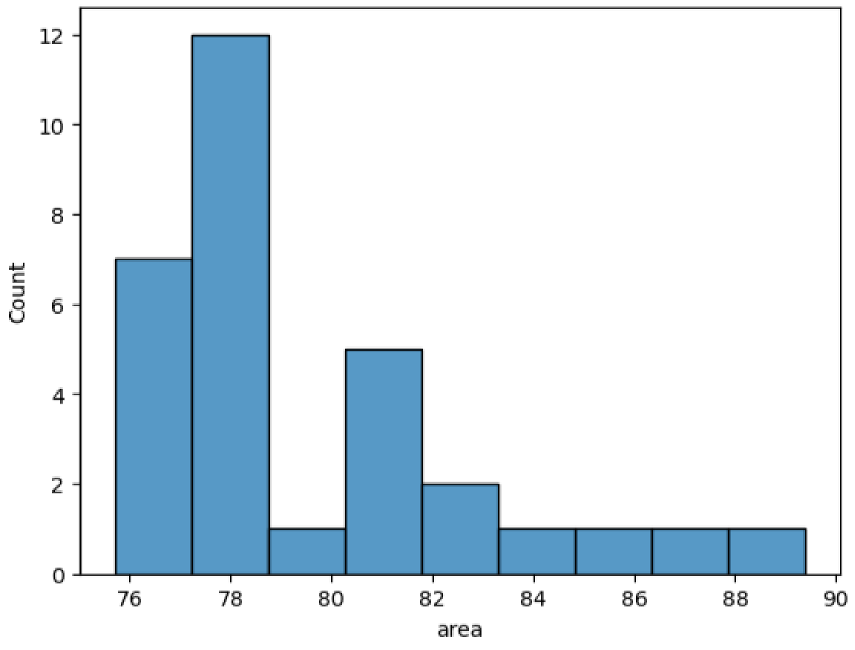


Figure A10. Histogram plot of the areas for Nest on the server side using the employees payload.

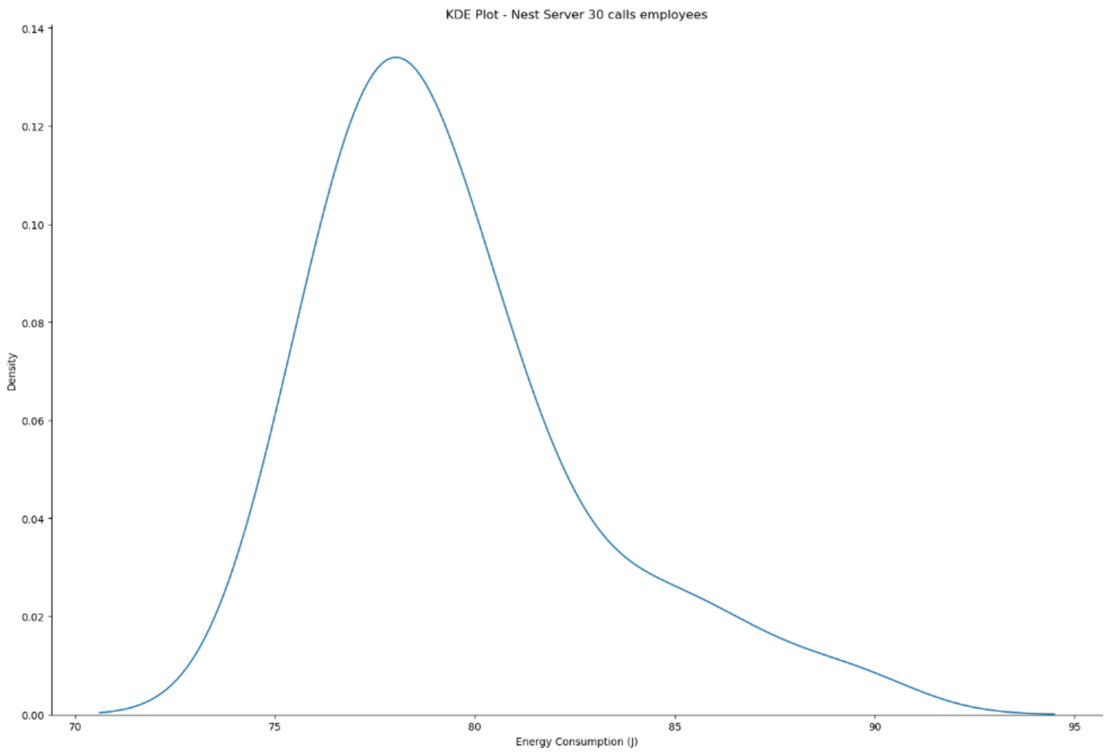


Figure A11. KDE plot of the areas for Nest on the server side using the employees payload.

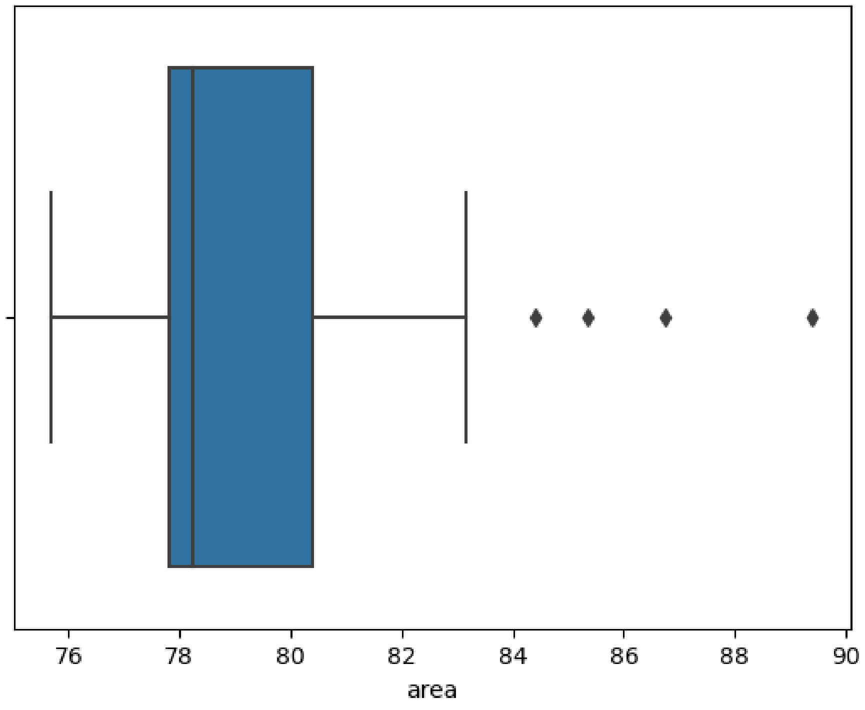


Figure A12. Box plot of the areas for Nest on the server side using the employees payload.

9.4.3. Nest Server Departments

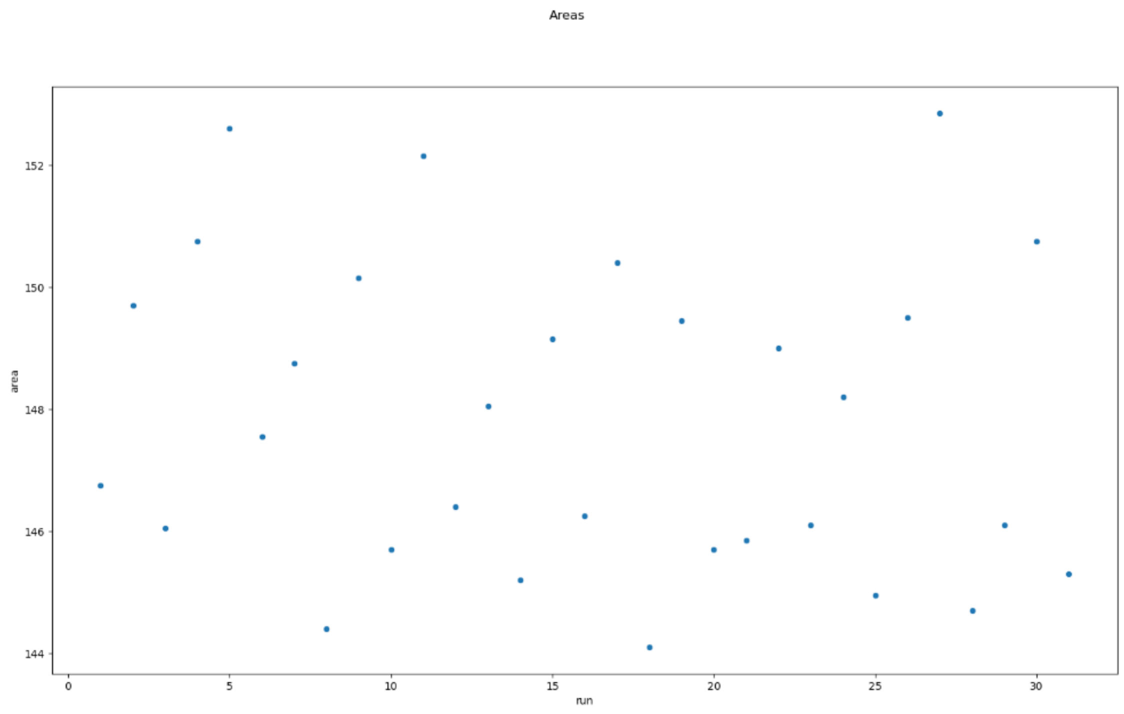


Figure A13. Scatter plot of the areas for Nest on the server side using the departments payload.

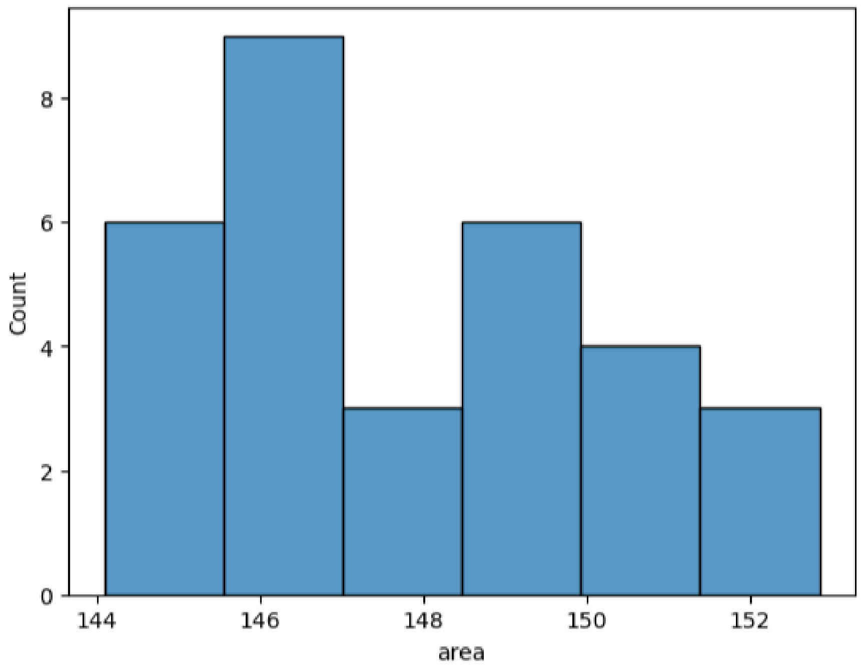


Figure A14. Histogram plot of the areas for Nest on the server side using the departments payload.

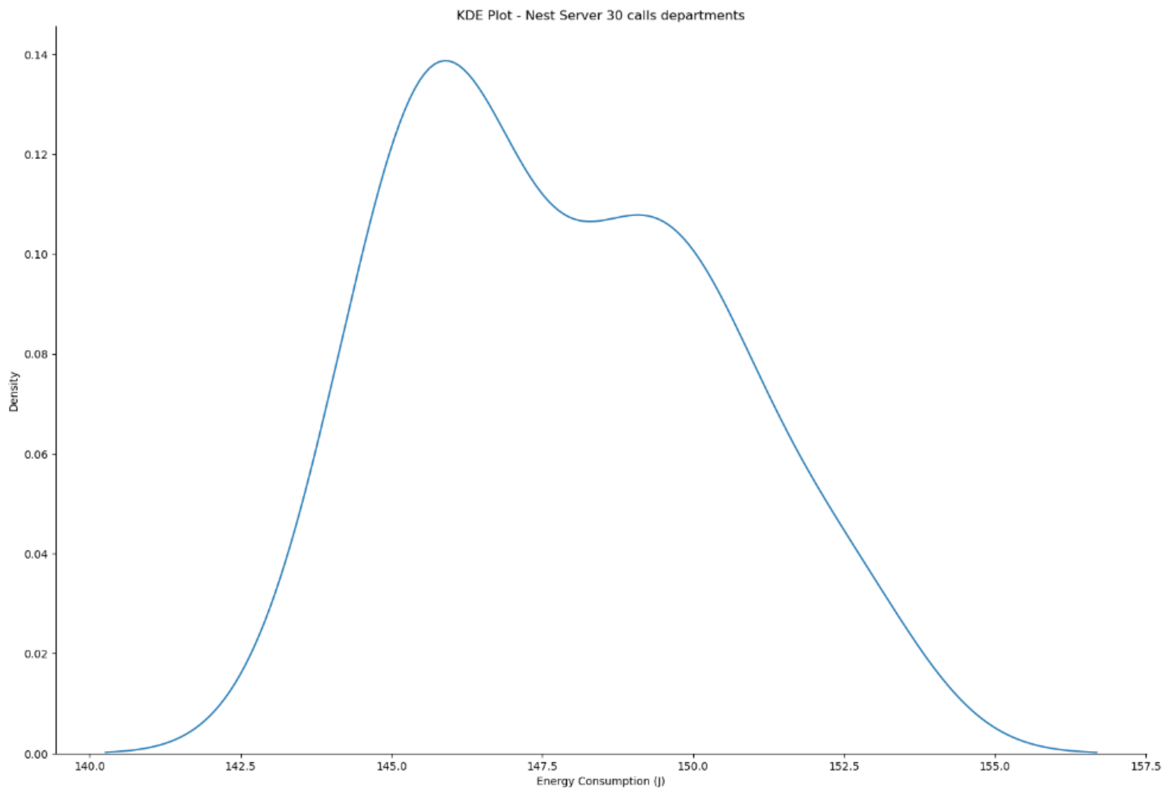


Figure A15. KDE plot of the areas for
Nest on the server side using the departments payload.

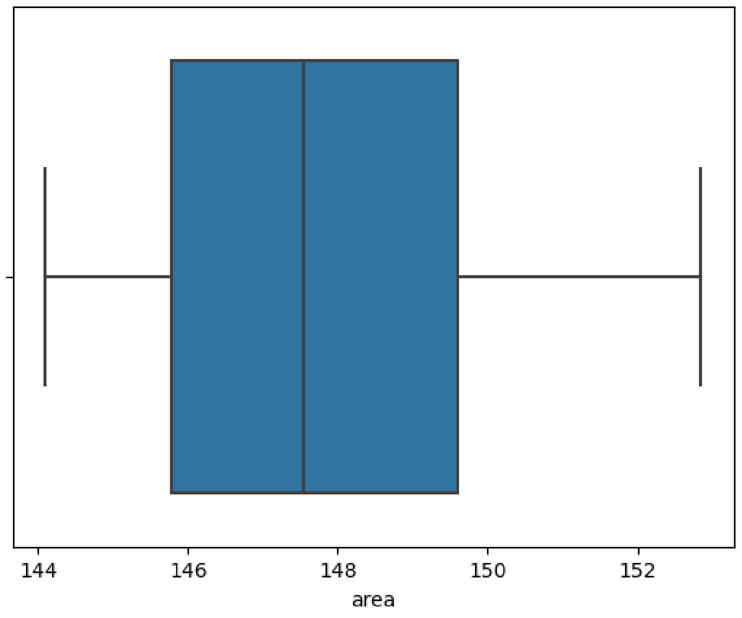


Figure A16. Box plot of the areas for
Nest on the server side using the departments payload.

9.4.4. Fastify Server Employees

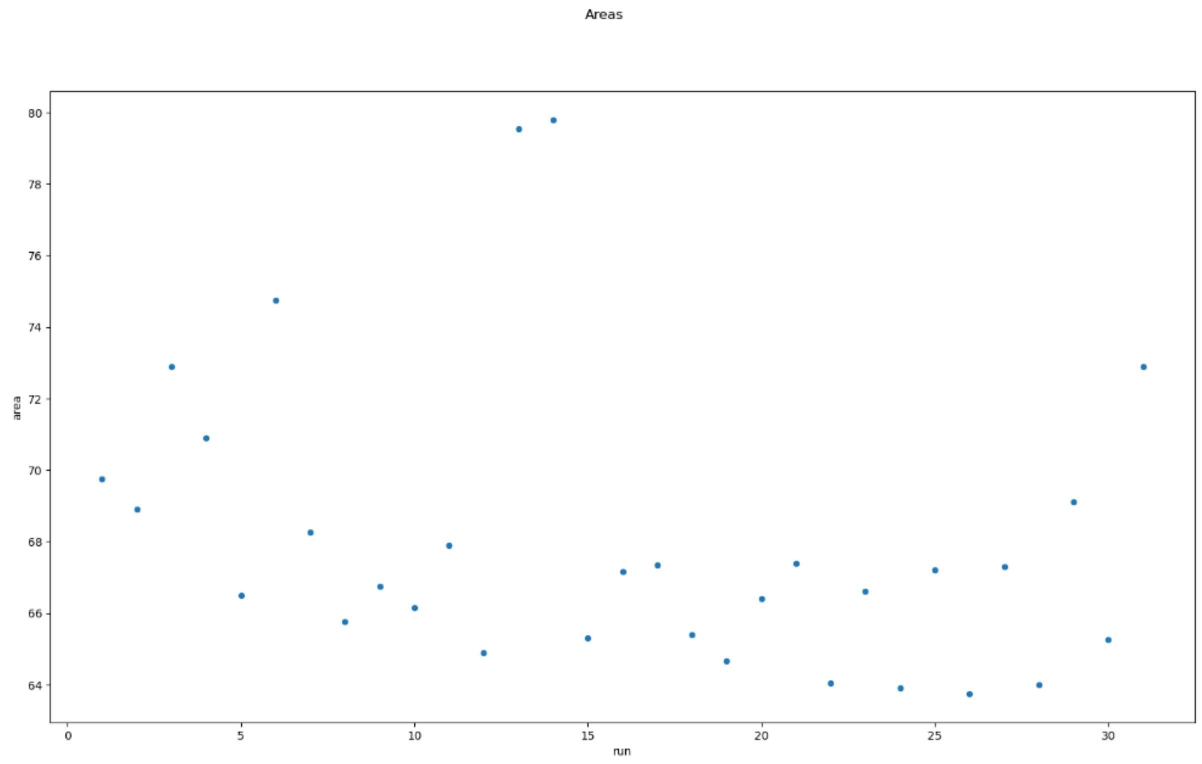


Figure A17. Scatter plot of the areas for Fastify on the server side using the employees payload.

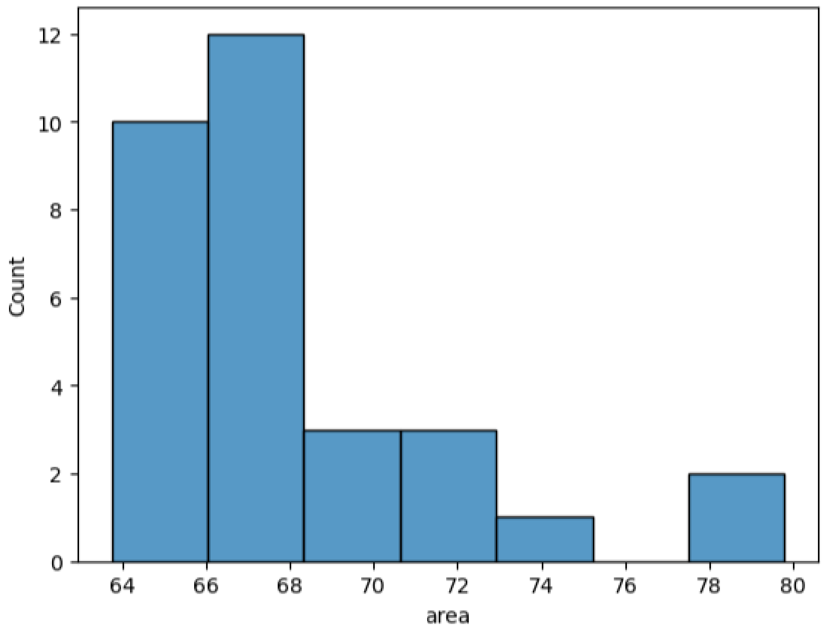


Figure A18. Histogram plot of the areas for Fastify on the server side using the employees payload.

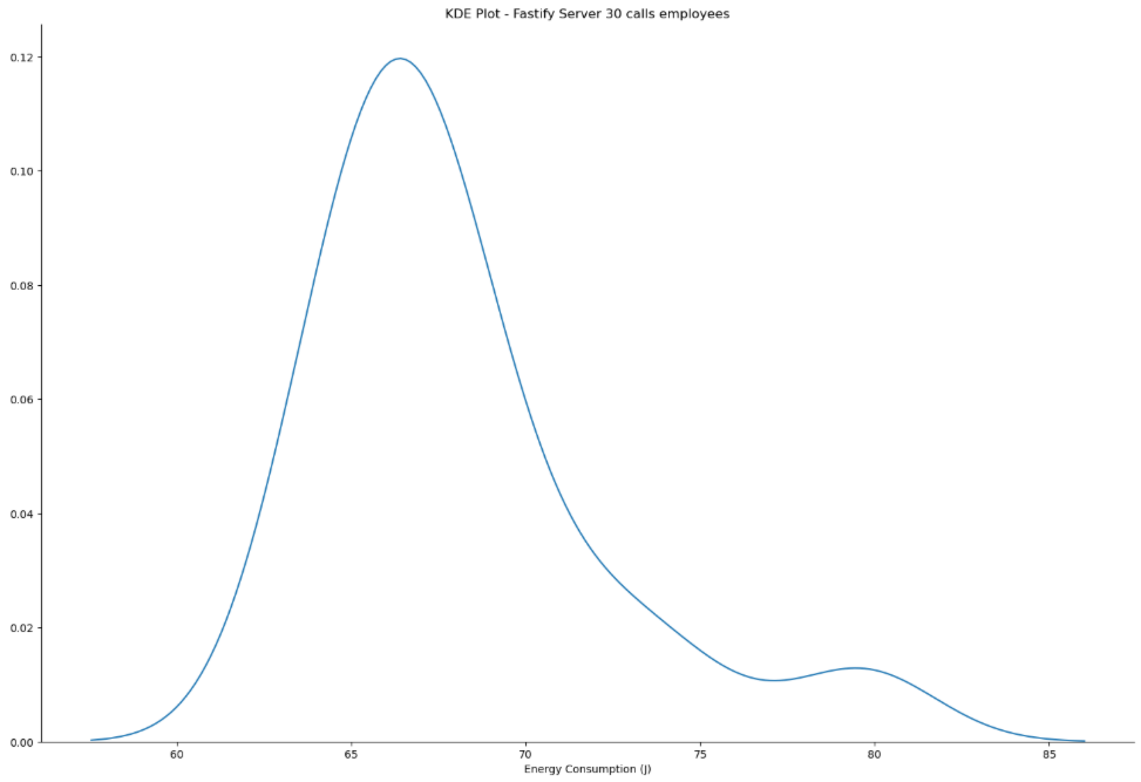


Figure A19. KDE plot of the areas for
Fastify on the server side using the employees payload.

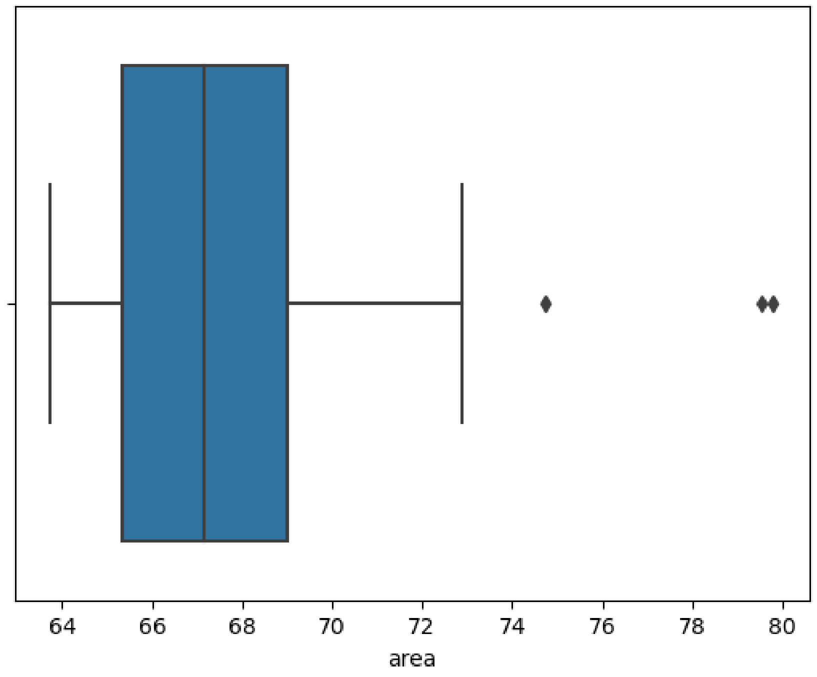


Figure A20. Box plot of the areas for
Fastify on the server side using the employees payload.

9.4.5. Fastify Server Departments

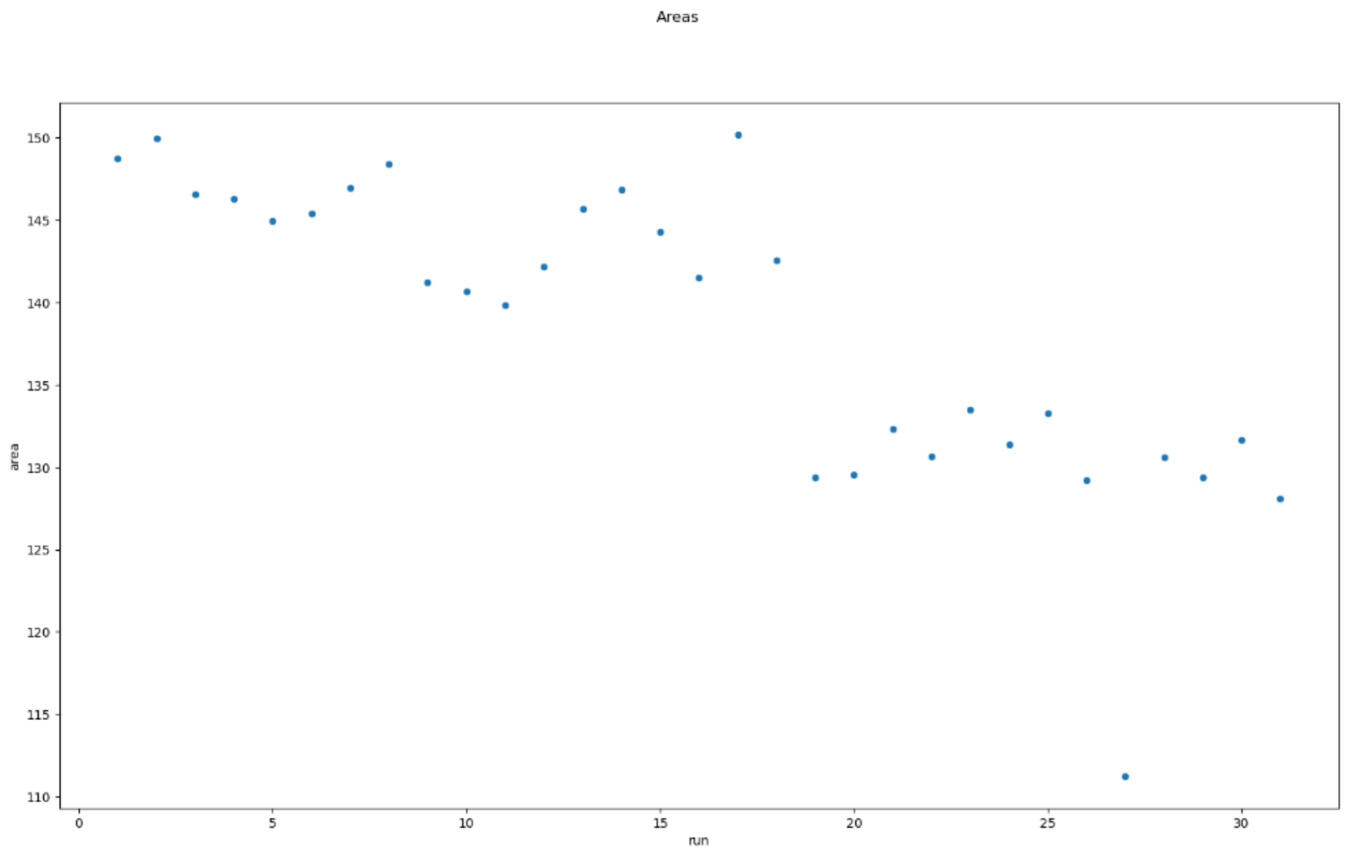


Figure A21. Scatter plot of the areas for Fastify on the server side using the departments payload.

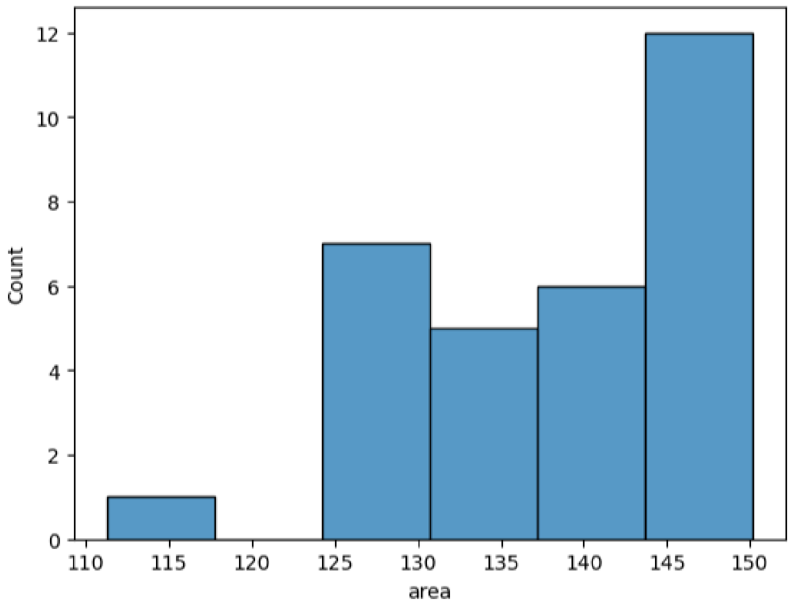


Figure A22. Histogram plot of the areas for Fastify on the server side using the departments payload.

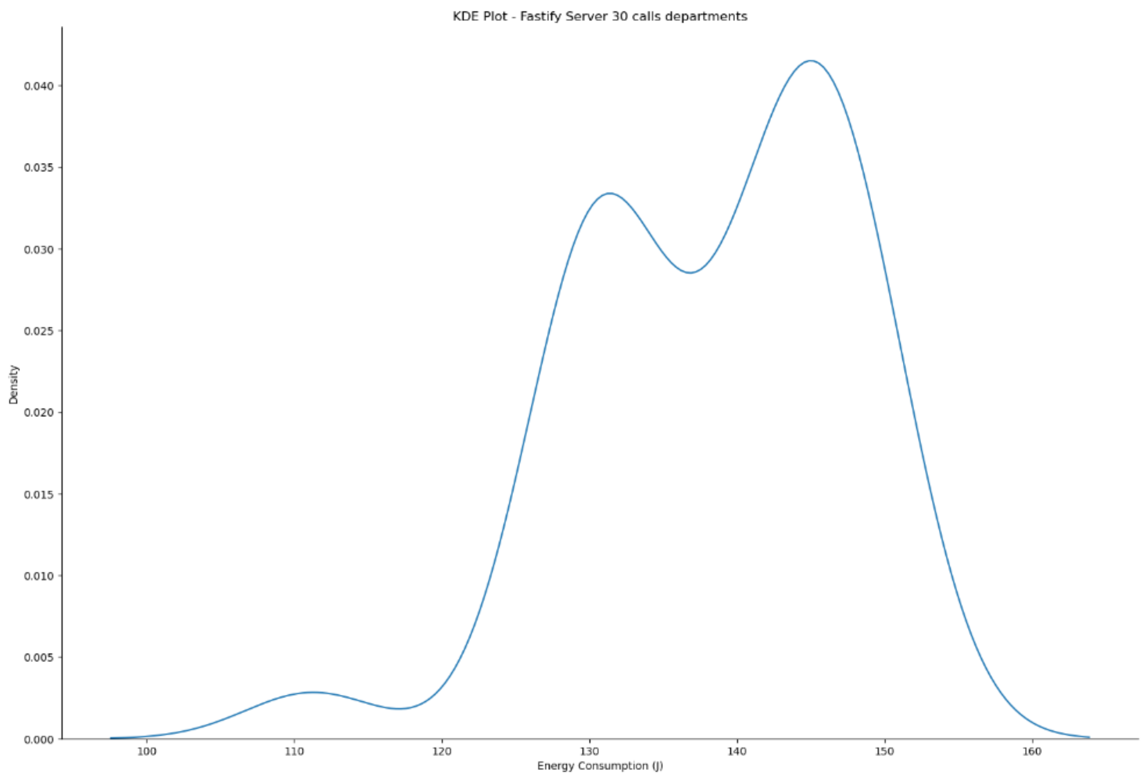


Figure A23. KDE plot of the areas for
Fastify on the server side using the departments payload.

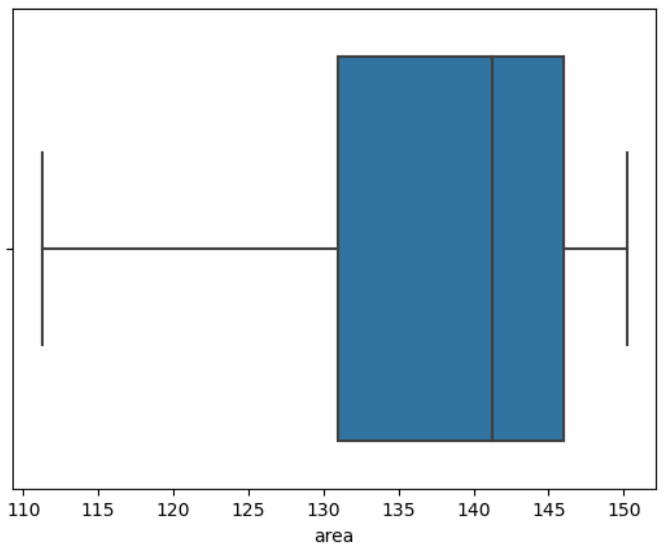


Figure A24. Box plot of the areas for
Fastify on the server side using the departments payload.

9.4.6. Koa Server Employees

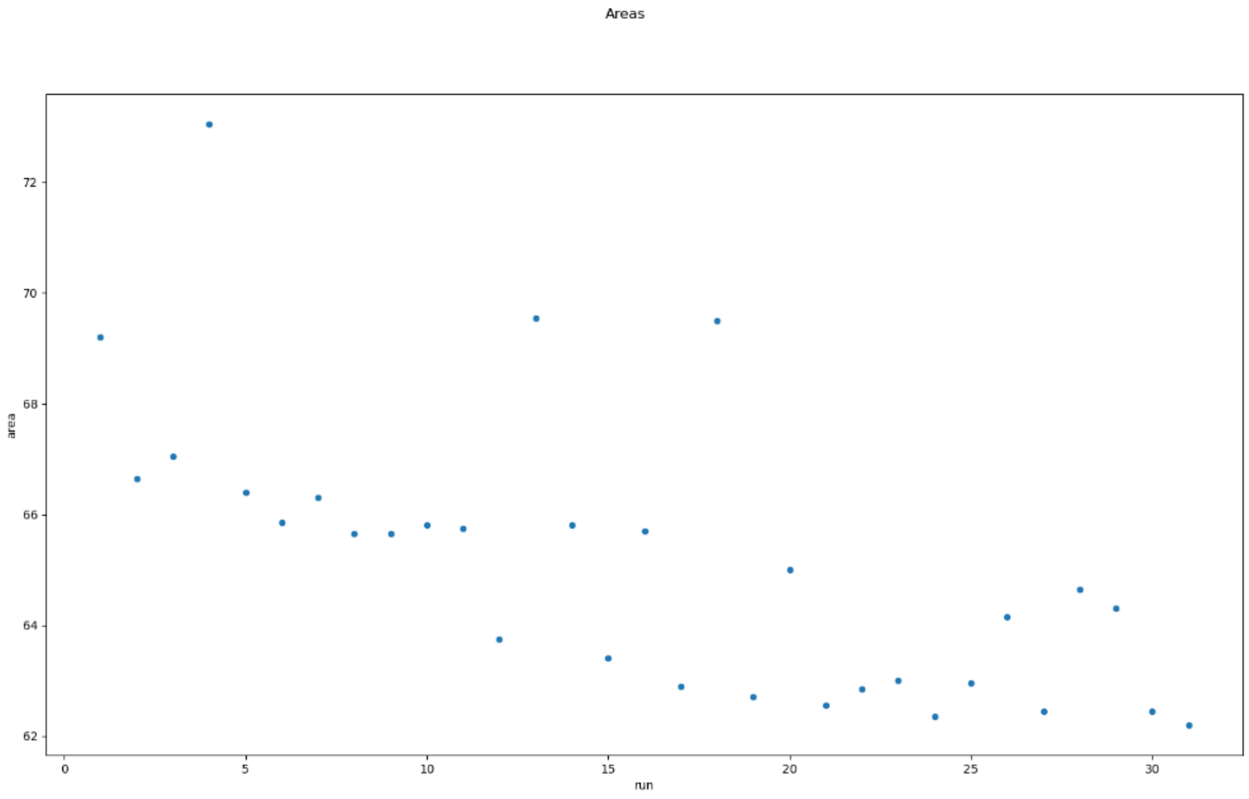


Figure A25. Scatter plot of the areas for Koa on the server side using the employees payload.

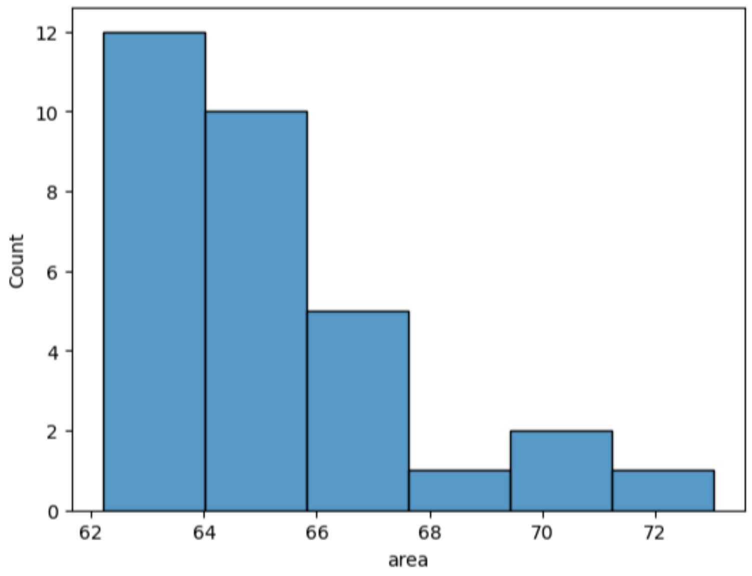


Figure A26. Histogram plot of the areas for Koa on the server side using the employees payload.

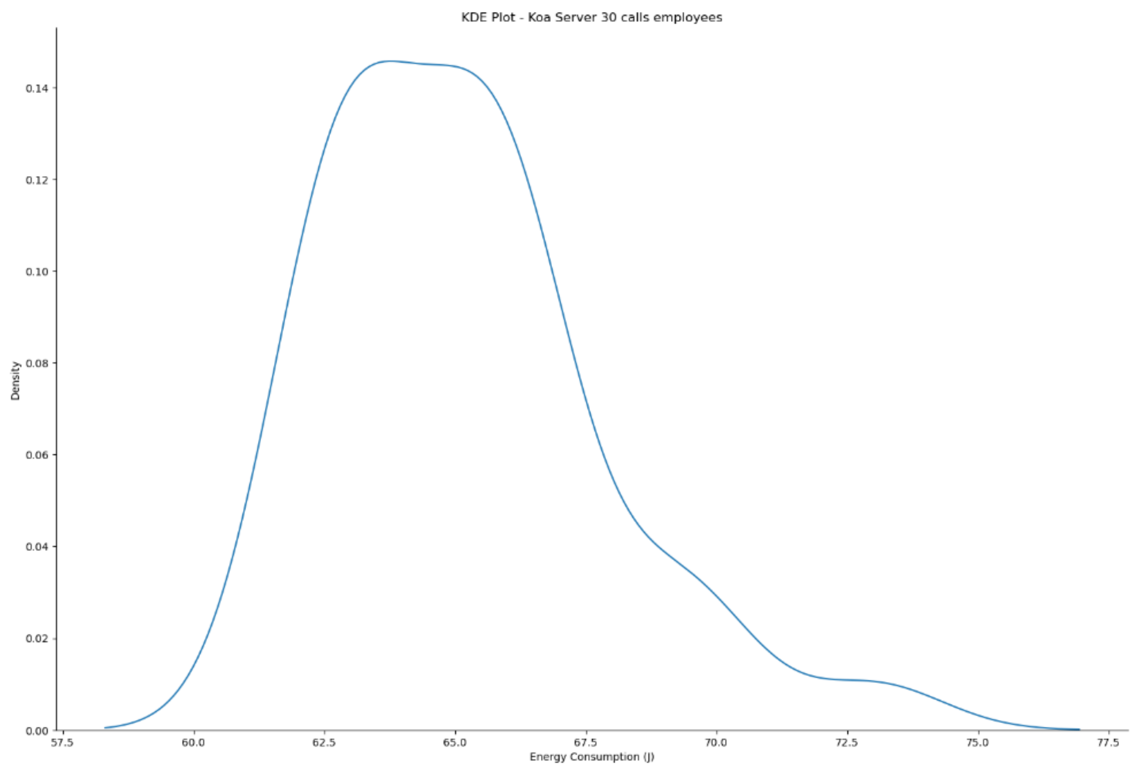


Figure A27. KDE plot of the areas for
Koa on the server side using the employees payload.

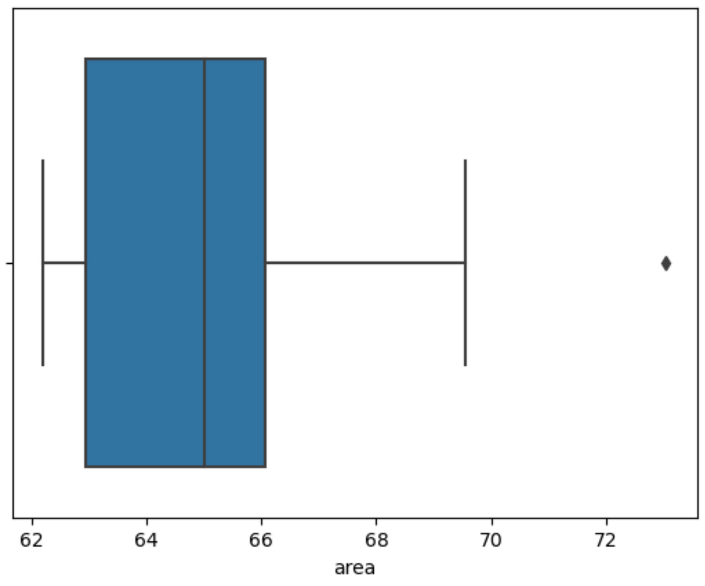


Figure A28. Box plot of the areas for
Koa on the server side using the employees payload.

9.4.7. Koa Server Departments

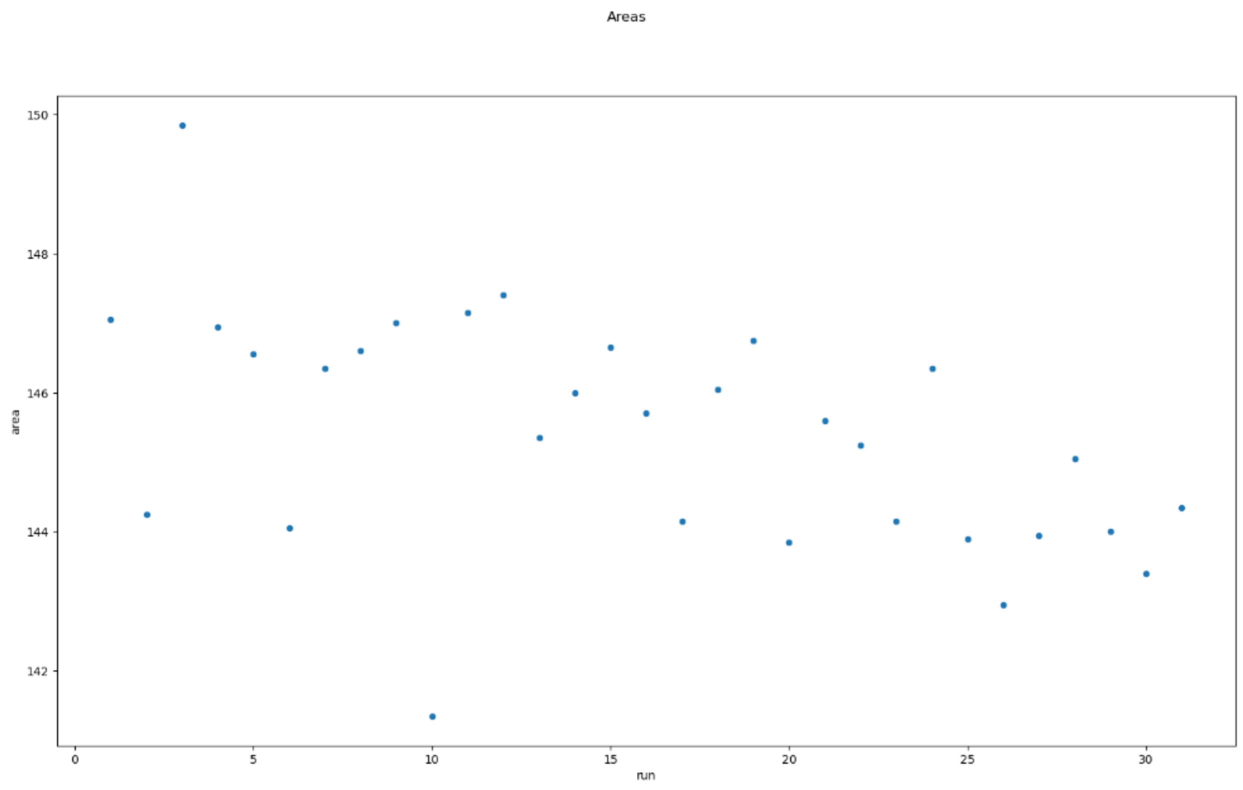


Figure A29. Scatter plot of the areas for Koa on the server side using the departments payload.

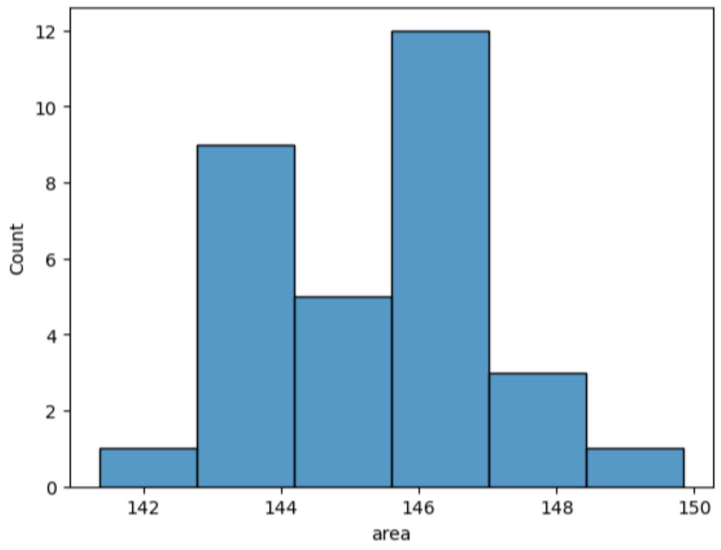


Figure A30. Histogram plot of the areas for Koa on the server side using the departments payload.

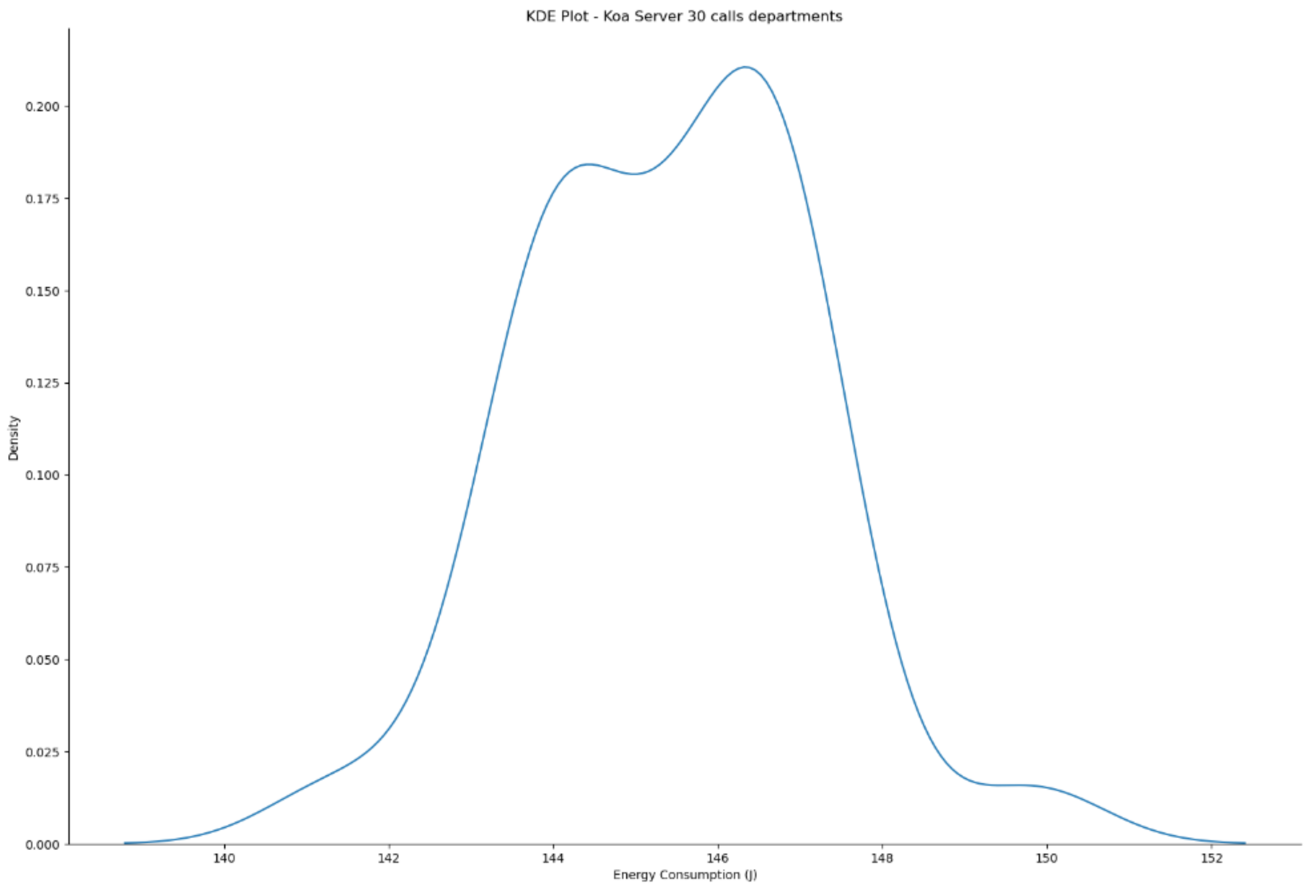


*Figure A31. KDE plot of the areas for
Koa on the server side using the departments payload.*

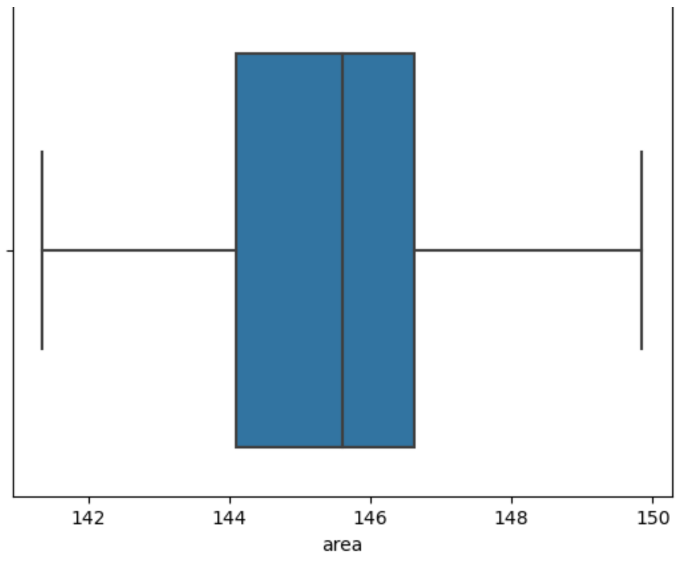


*Figure A32. Box plot of the areas for
Koa on the server side using the departments payload.*

9.4.8. Express Client Employees

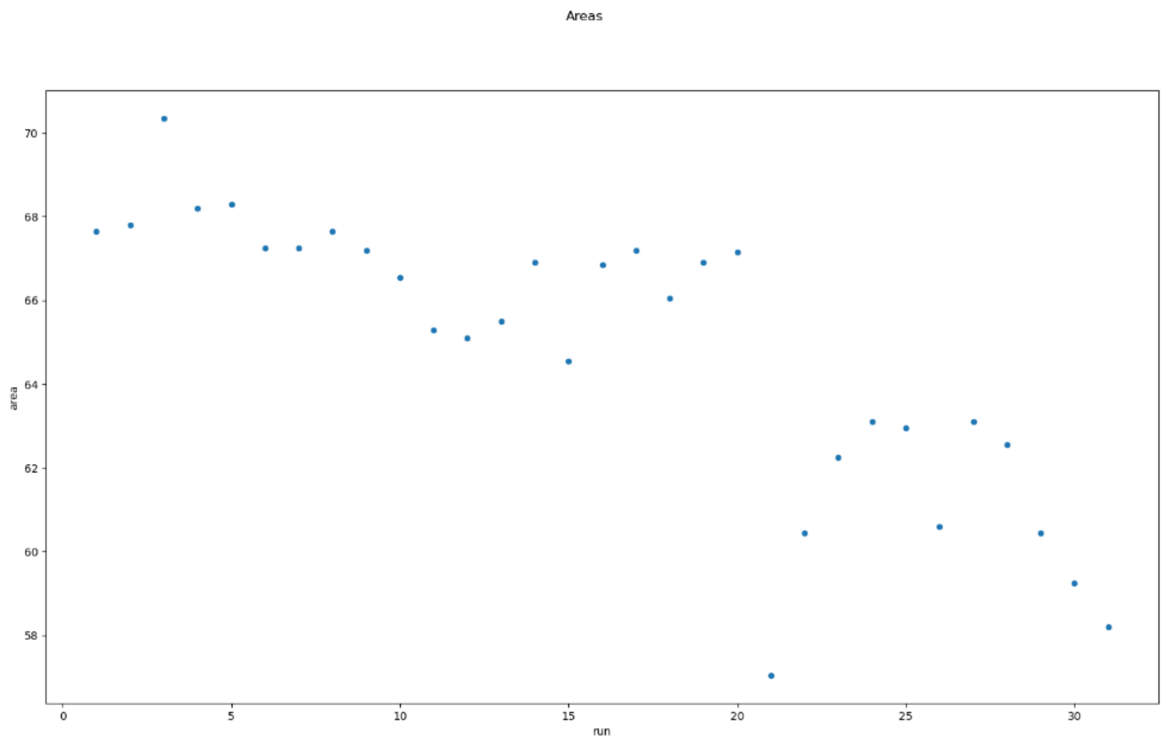


Figure A33. Scatter plot of the areas for Express on the client side using the employees payload.

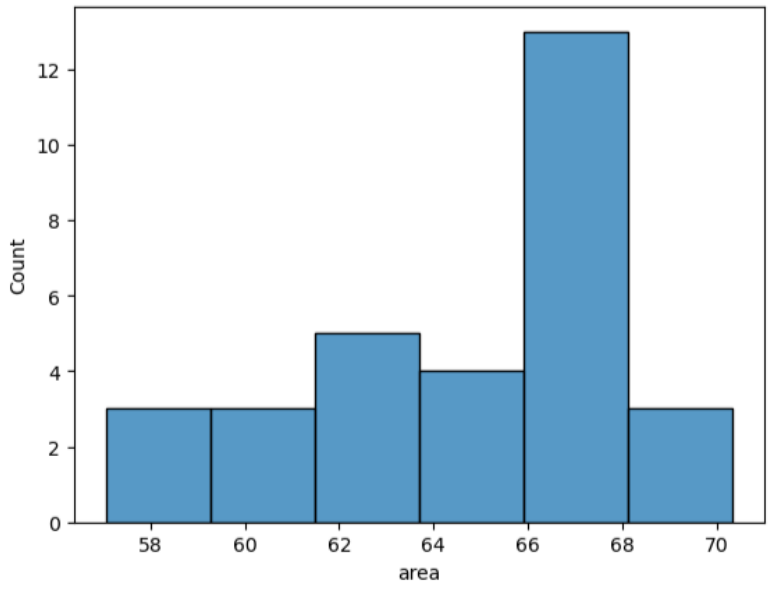


Figure A34. Histogram plot of the areas for Express on the client side using the employees payload.

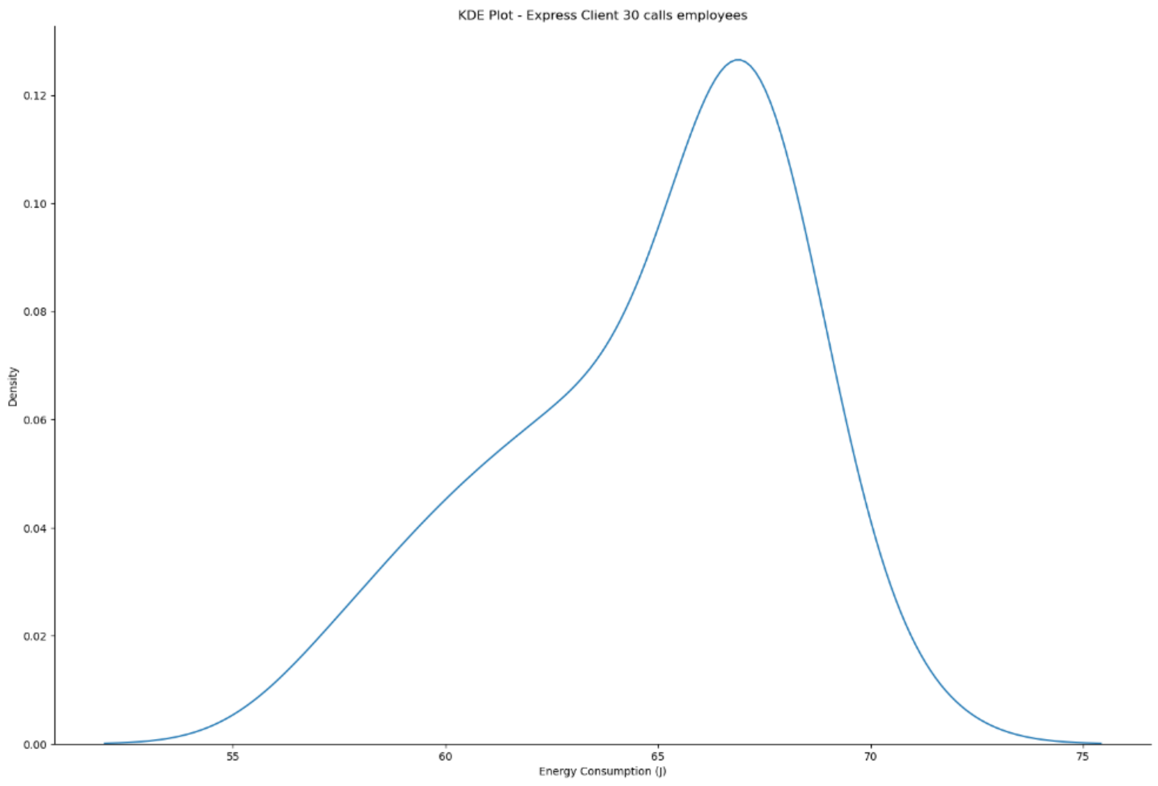


Figure A35. KDE plot of the areas for
Express on the client side using the employees payload.

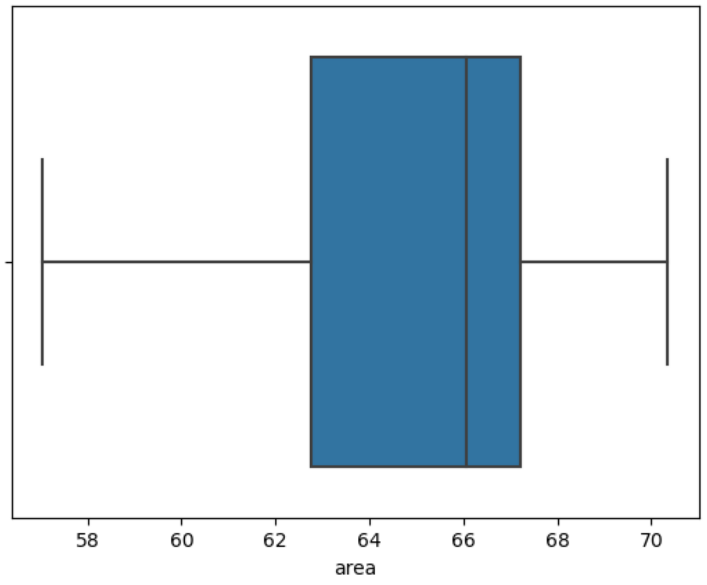


Figure A36. Box plot of the areas for
Express on the client side using the employees payload.

9.4.9. Express Client Departments

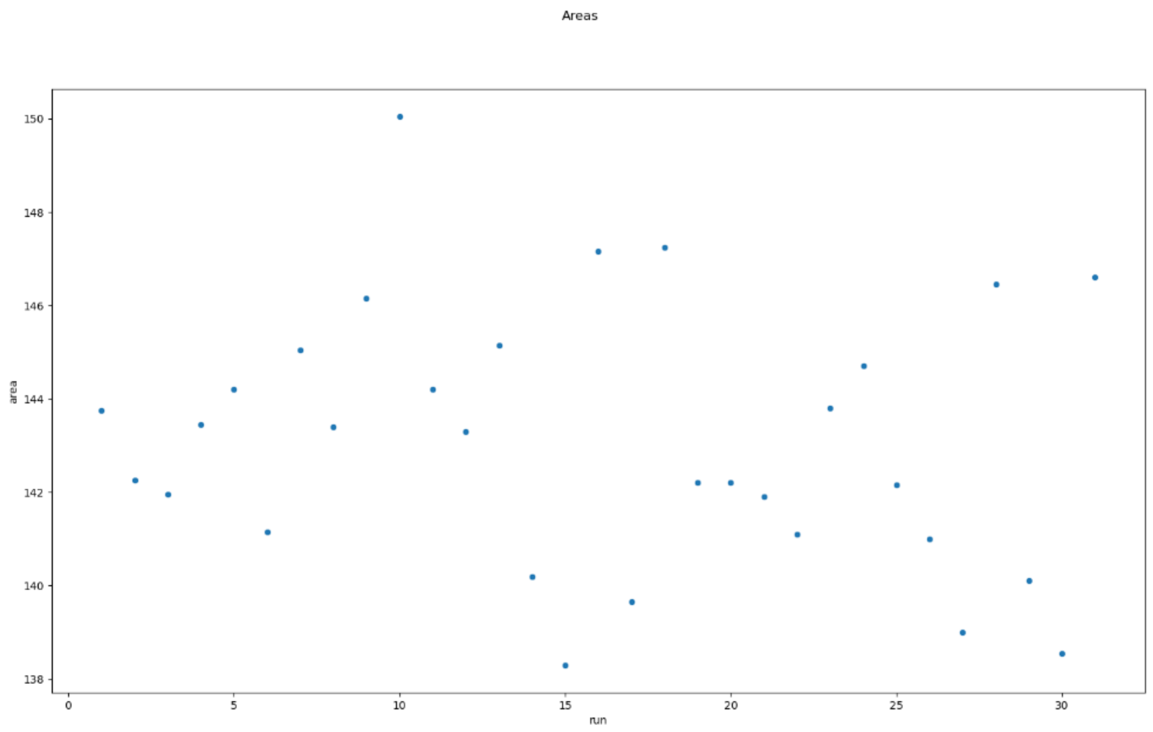


Figure A37. Scatter plot of the areas for Express on the client side using the departments payload.

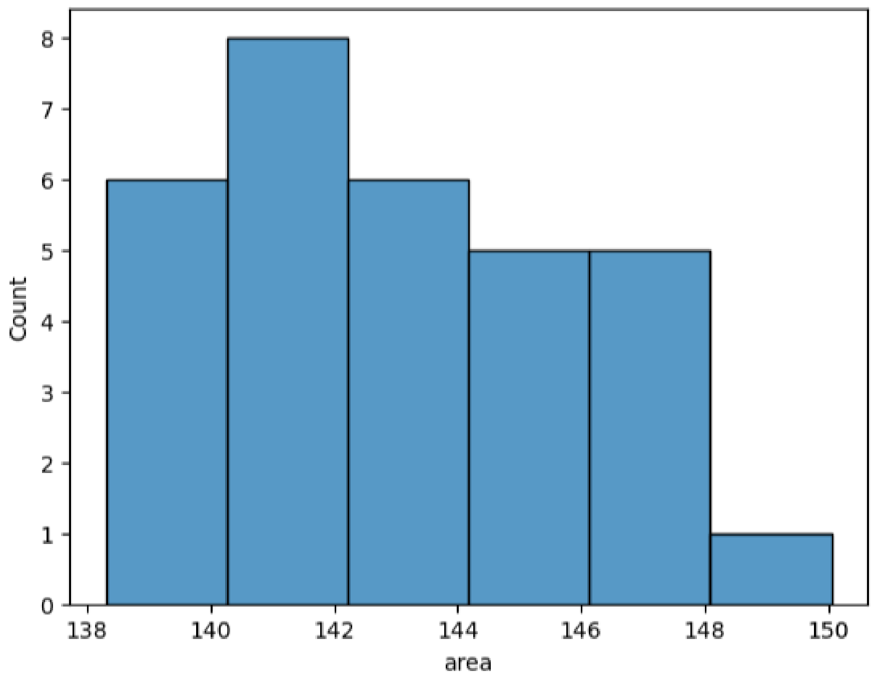


Figure A38. Histogram plot of the areas for Express on the client side using the departments payload.

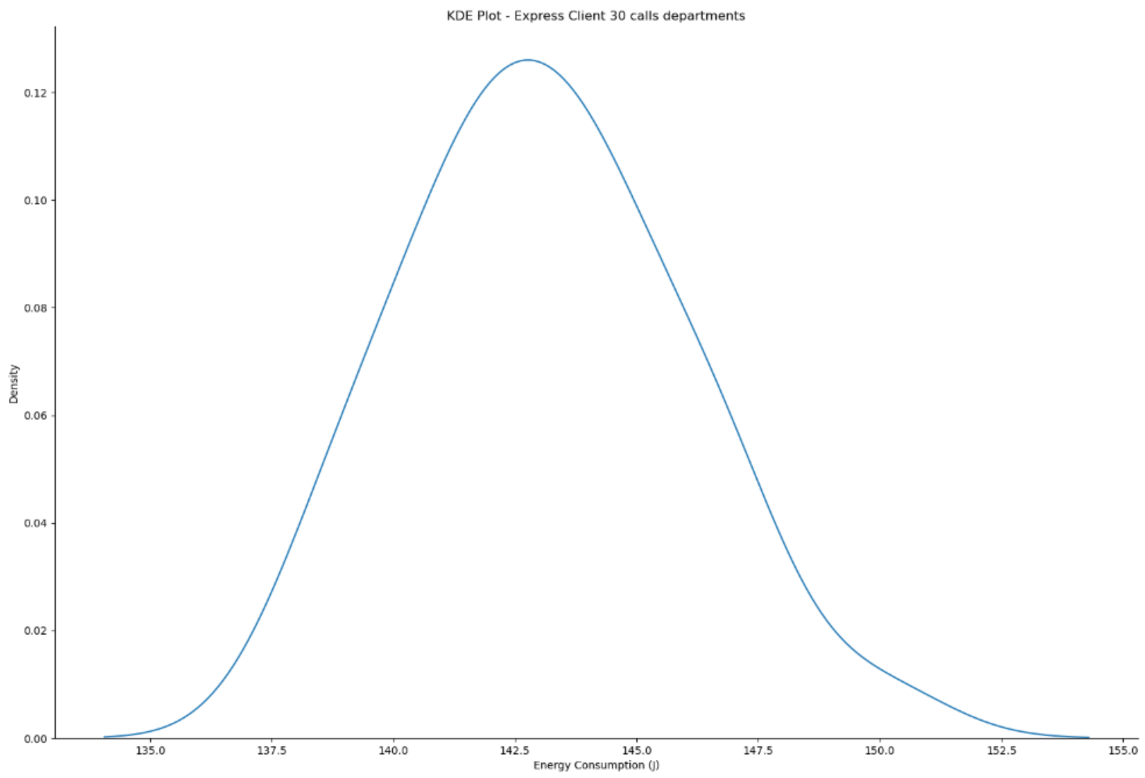


Figure A39. KDE plot of the areas for
Express on the client side using the departments payload.

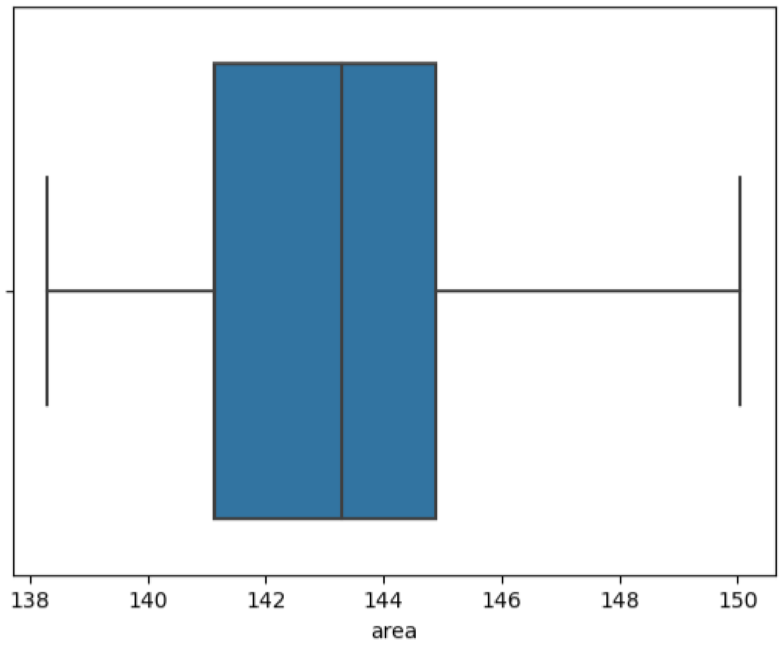


Figure A40. Box plot of the areas for
Express on the client side using the departments payload.

9.4.10. Nest Client Employees

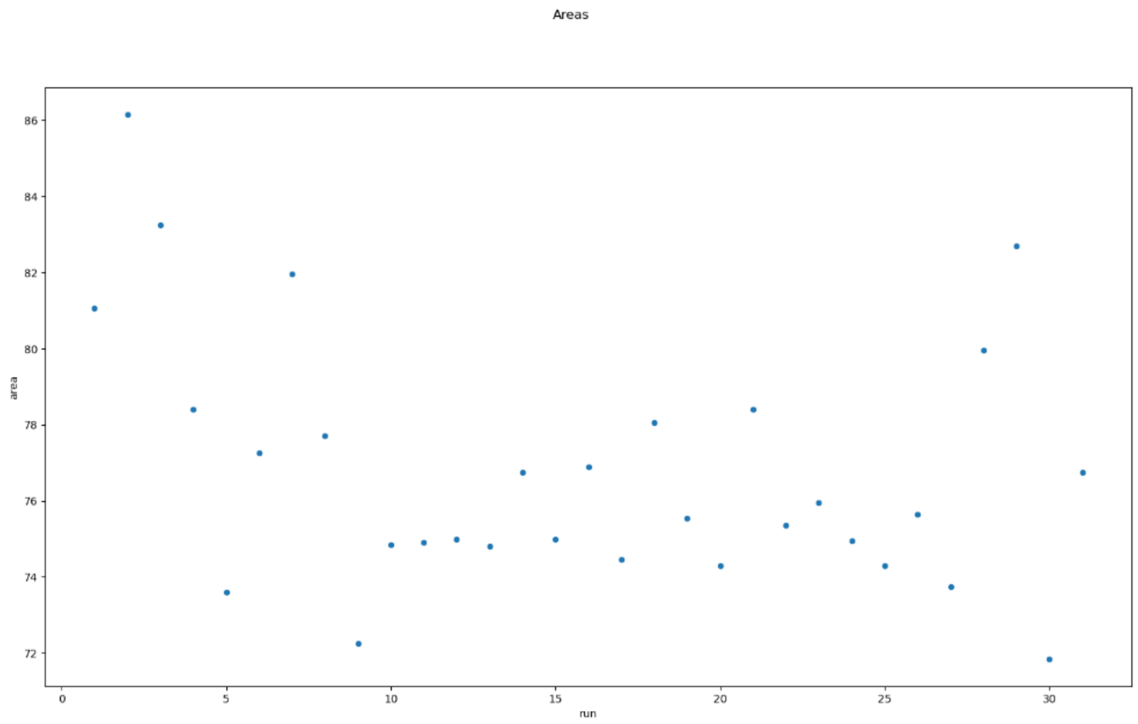


*Figure A41. Scatter plot of the areas for
Nest on the client side using the employees payload.*

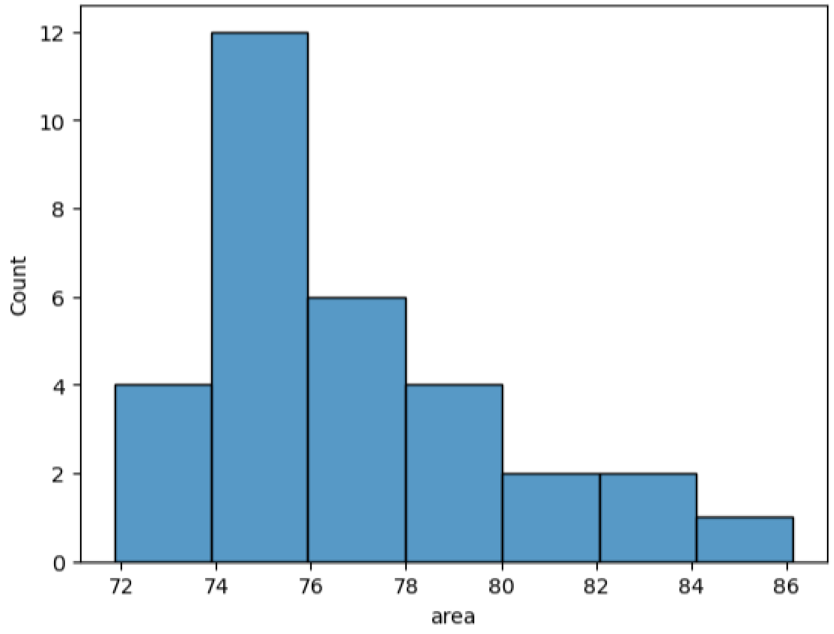


*Figure A42. Histogram plot of the areas for
Nest on the client side using the employees payload.*

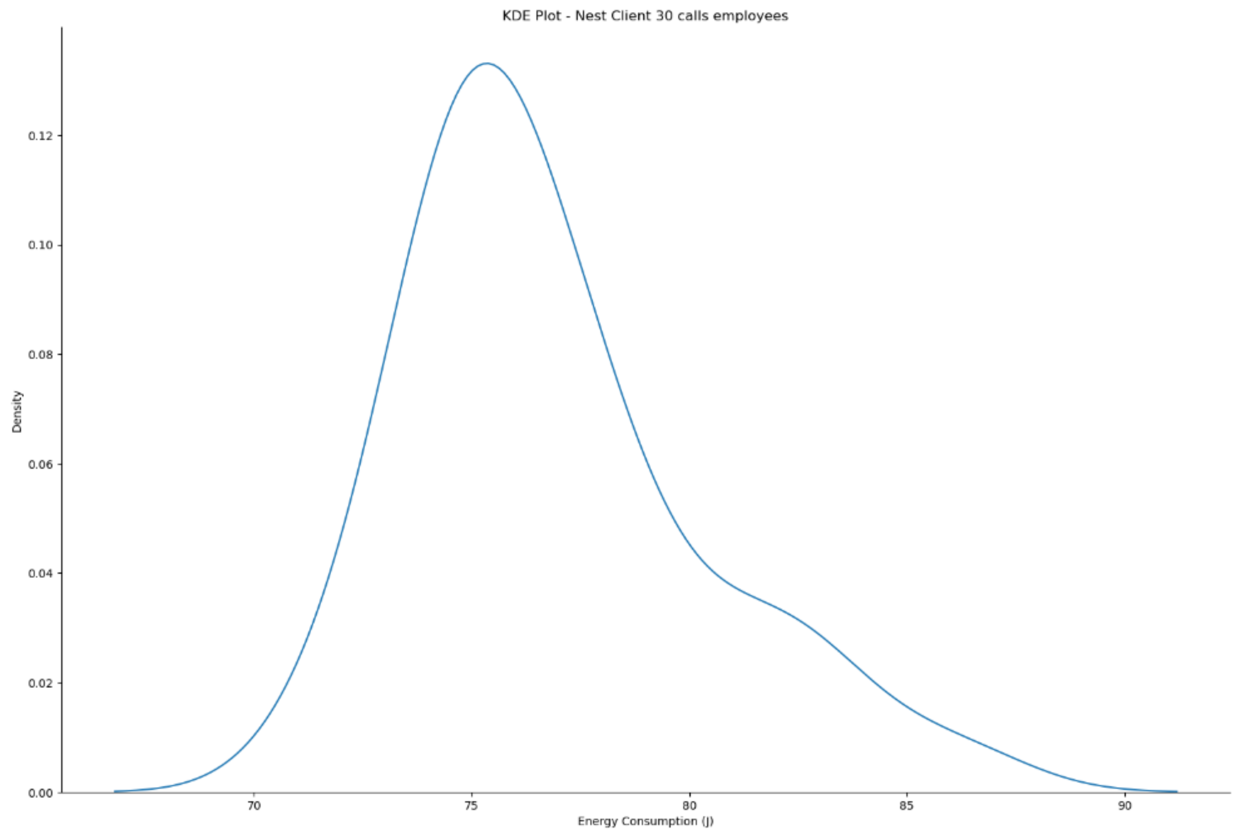


Figure A43. KDE plot of the areas for
Nest on the client side using the employees payload.

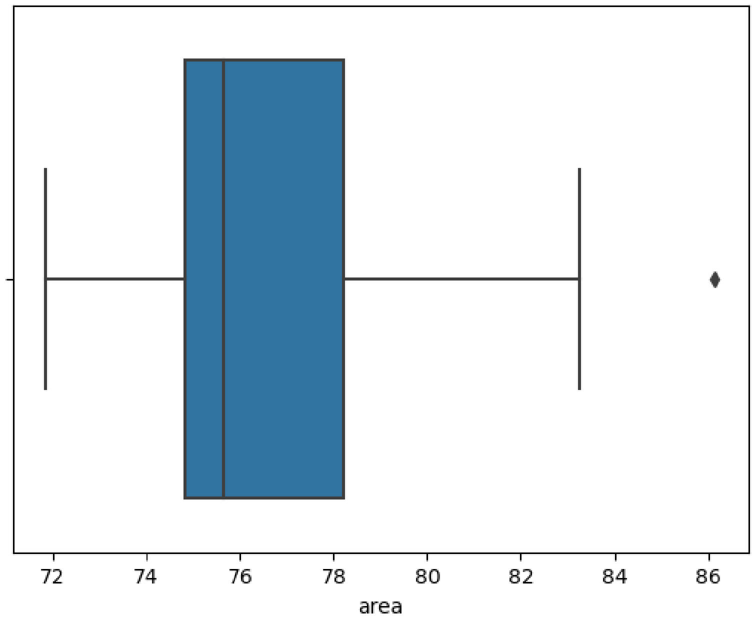


Figure A44. Box plot of the areas for
Nest on the client side using the employees payload.

9.4.11. Nest Client Departments

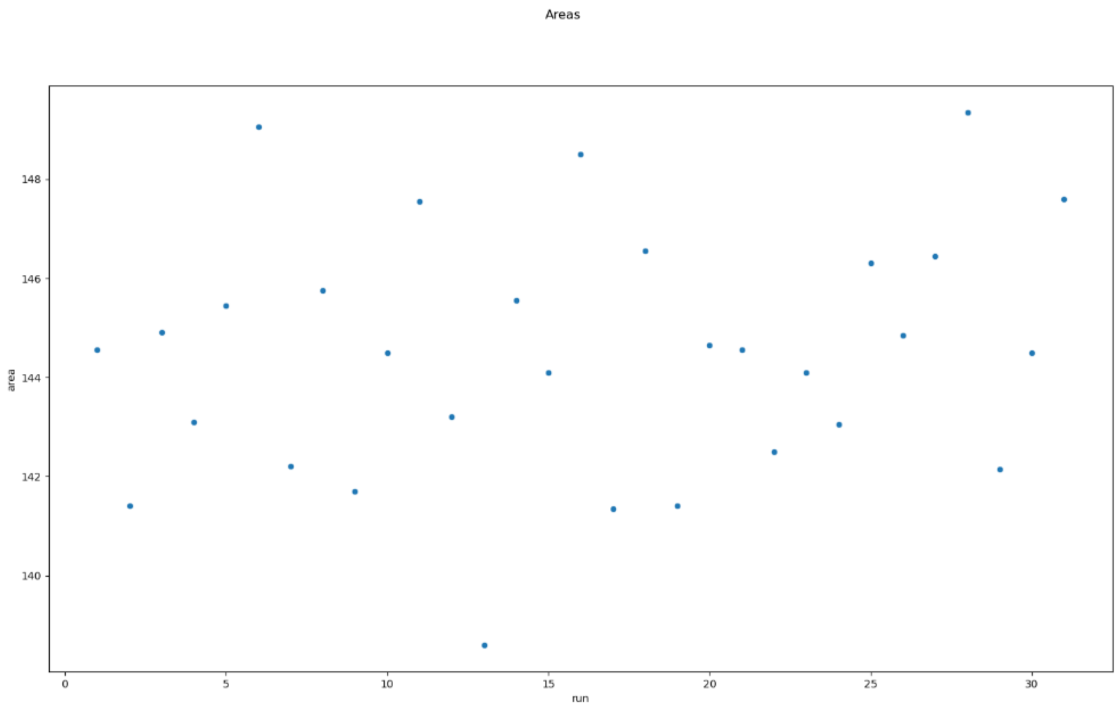


Figure A45. Scatter plot of the areas for Nest on the client side using the departments payload.

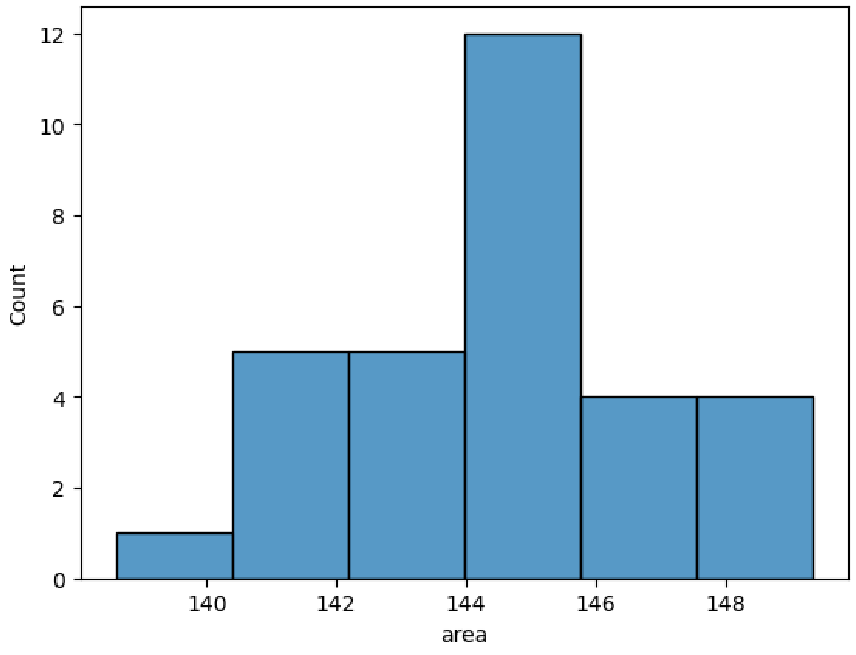


Figure A46. Histogram plot of the areas for Nest on the client side using the departments payload.

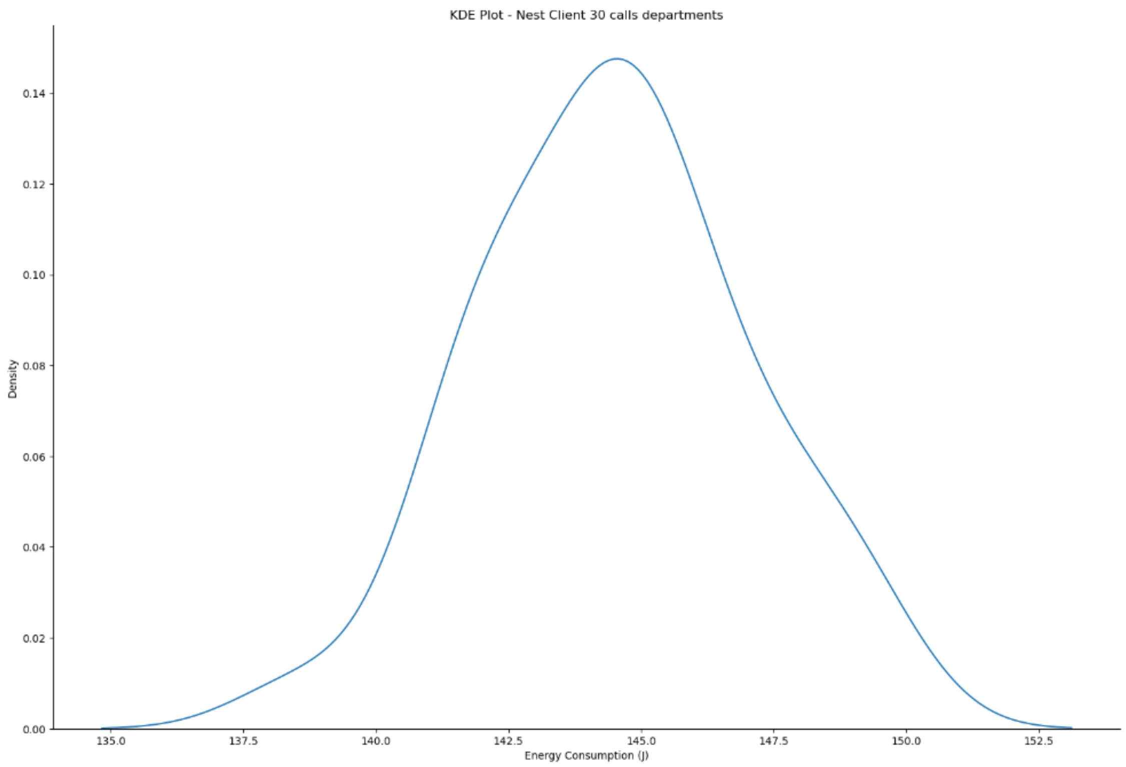


Figure A47. KDE plot of the areas for
Nest on the client side using the departments payload.

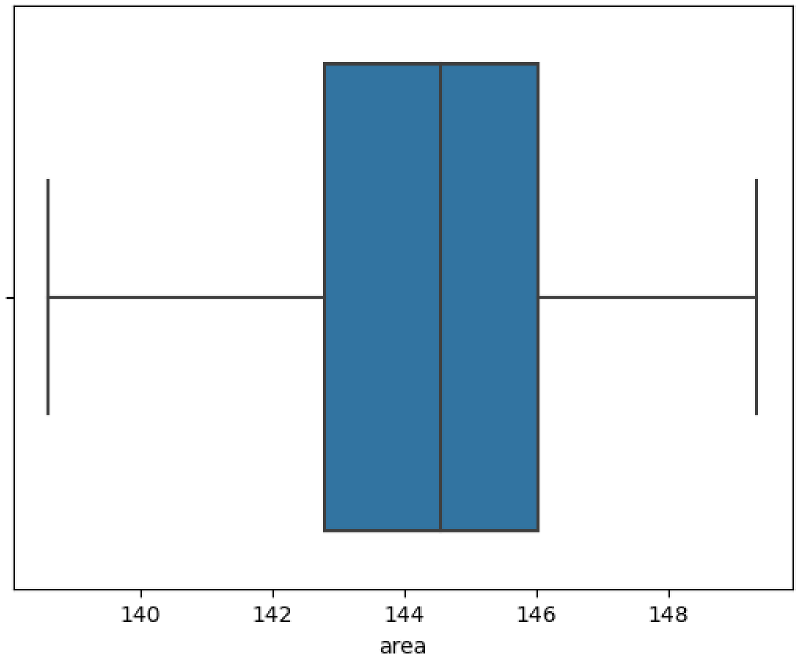


Figure A48. Box plot of the areas for
Nest on the client side using the departments payload.

9.4.12. Fastify Client Employees

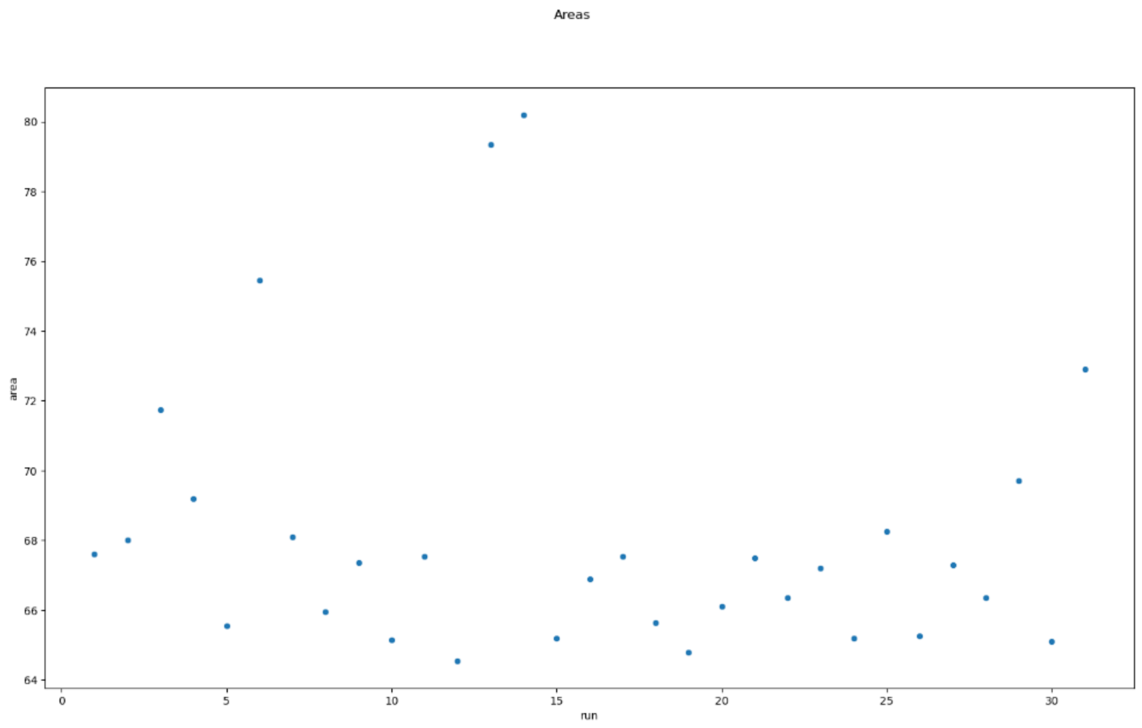


Figure A49. Scatter plot of the areas for Fastify on the client side using the employees payload.

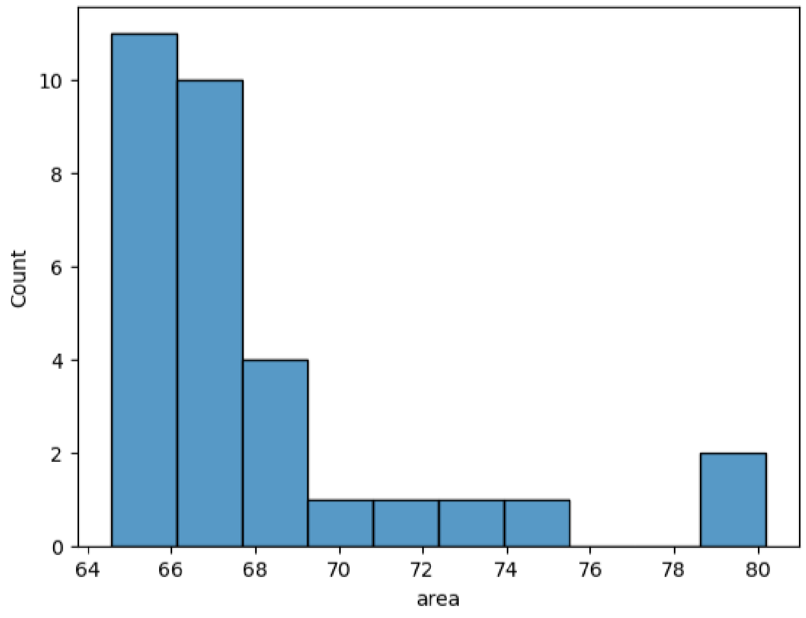


Figure A50. Histogram plot of the areas for Fastify on the client side using the employees payload.

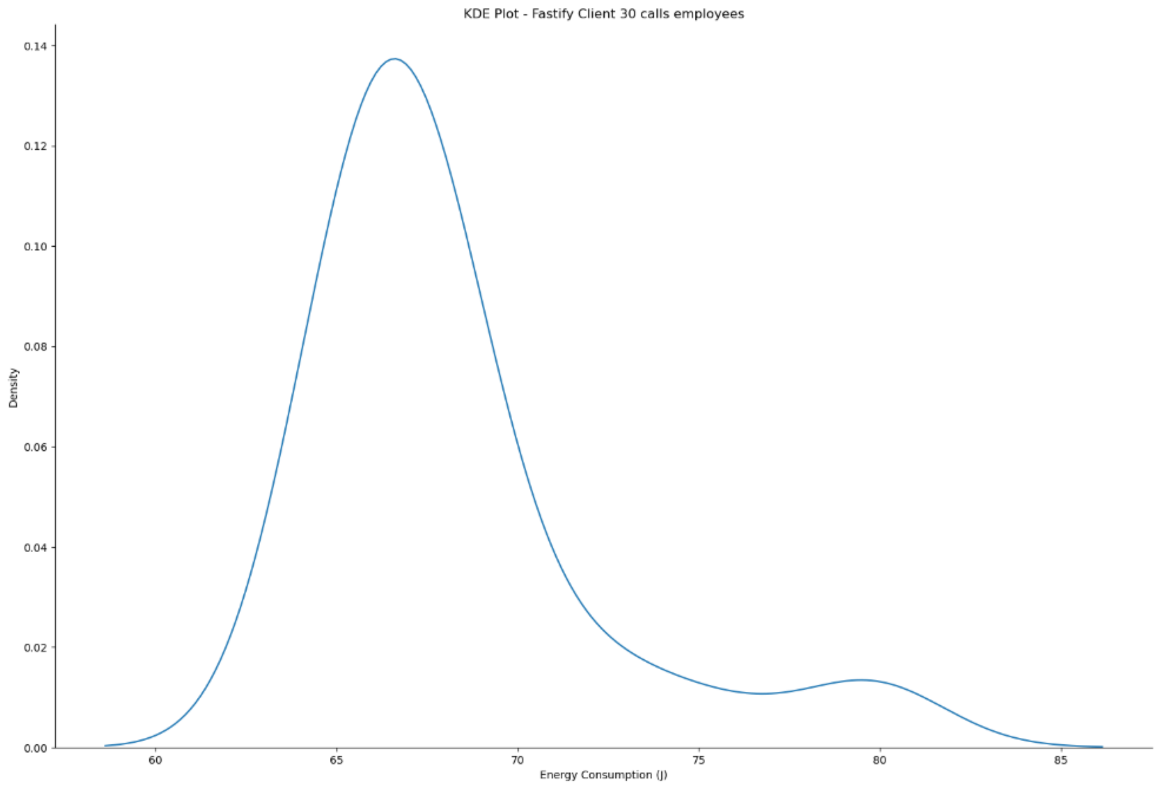


Figure A51. KDE plot of the areas for
Fastify on the client side using the employees payload.

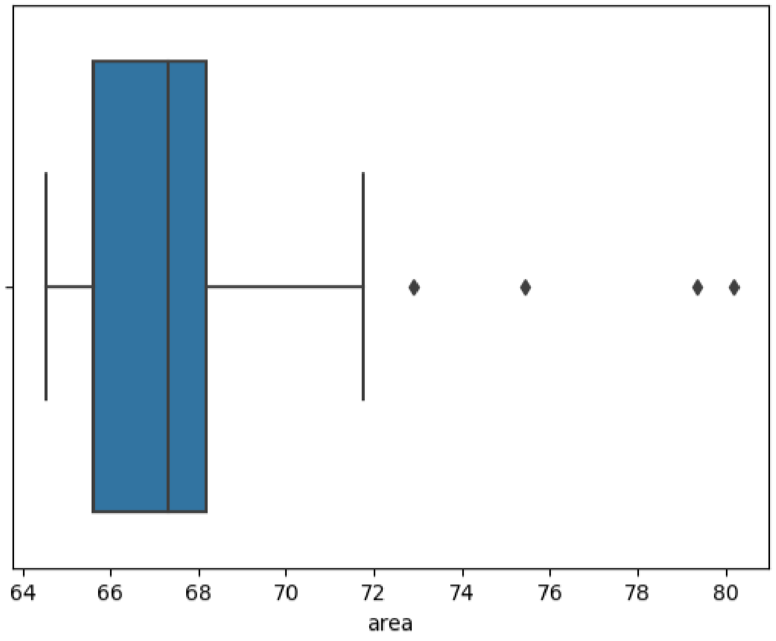


Figure A52. Box plot of the areas for
Fastify on the client side using the employee payload.

9.4.13. Fastify Client Departments

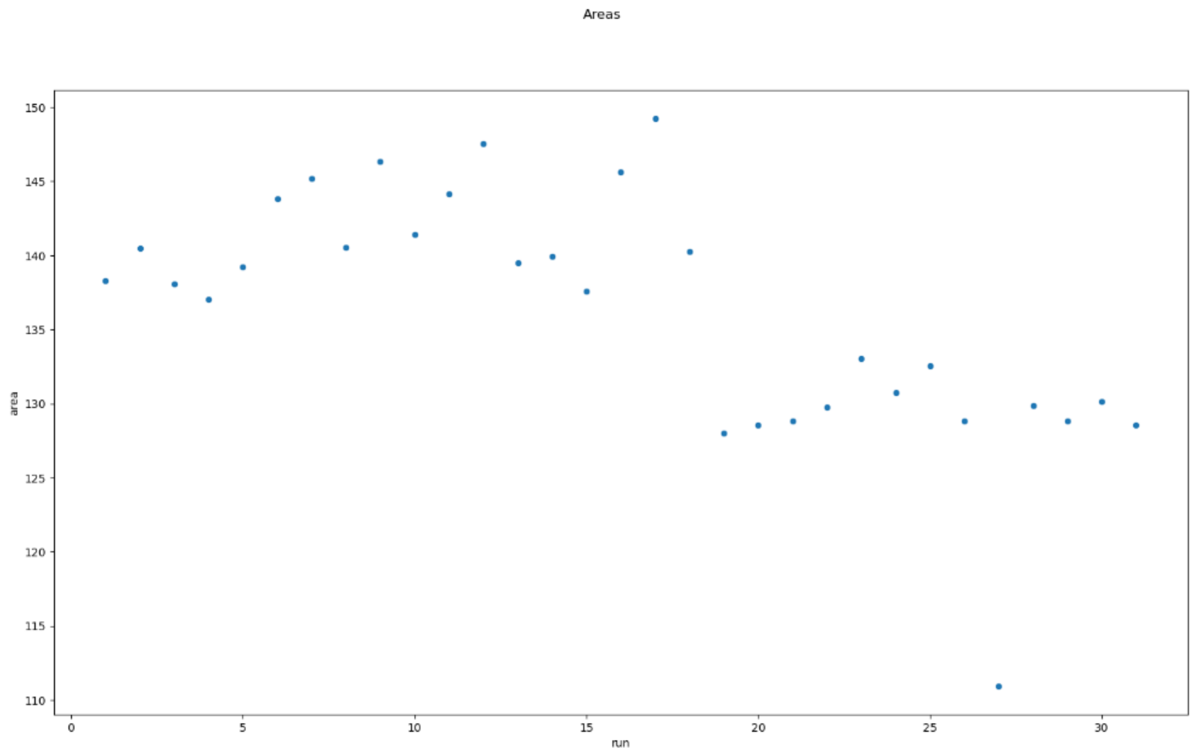


Figure A53. Scatter plot of the areas for Fastify on the client side using the departments payload.

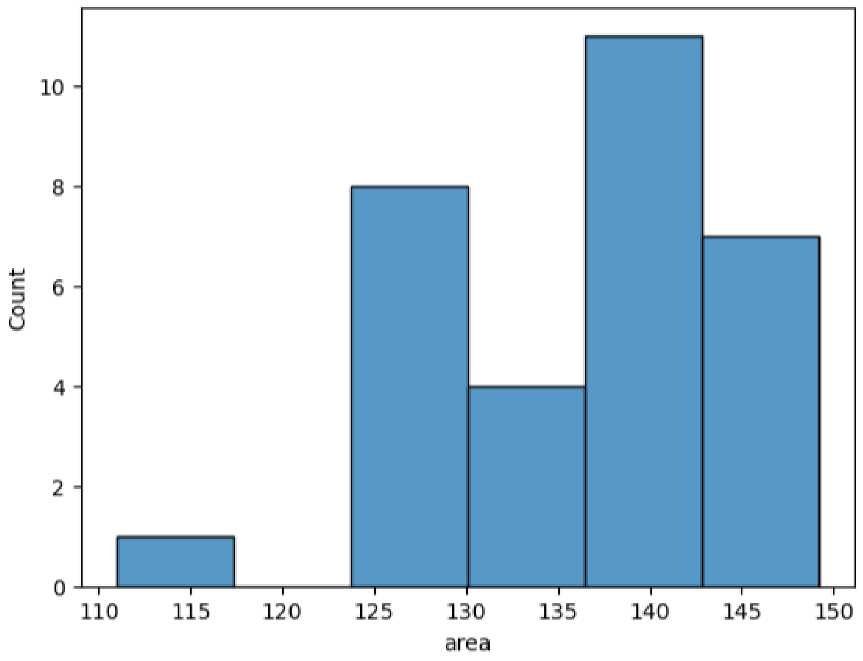


Figure A54. Histogram plot of the areas for Fastify on the client side using the departments payload.

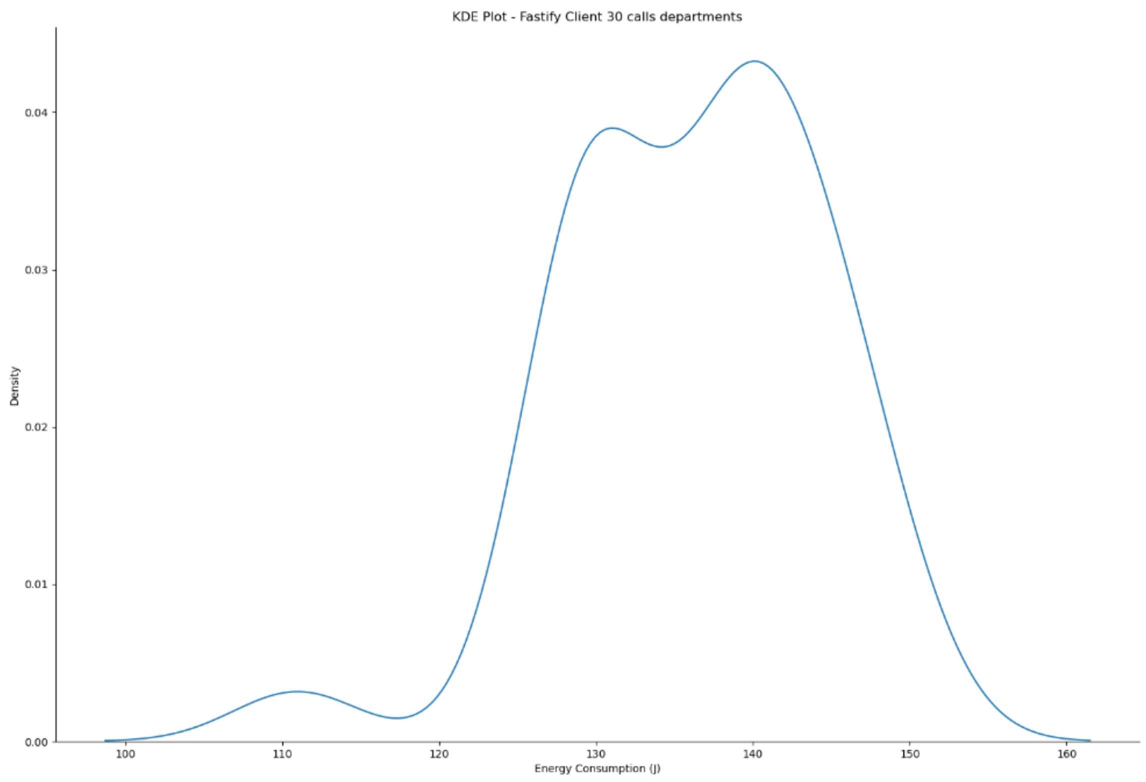


Figure A55. KDE plot of the areas for
Fastify on the client side using the departments payload.

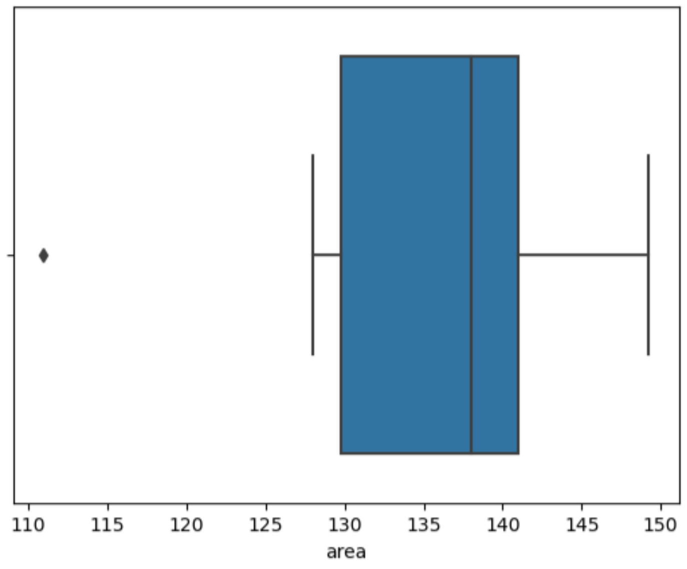


Figure A56. Box plot of the areas for
Fastify on the client side using the departments payload.

9.4.14. Koa Client Employees

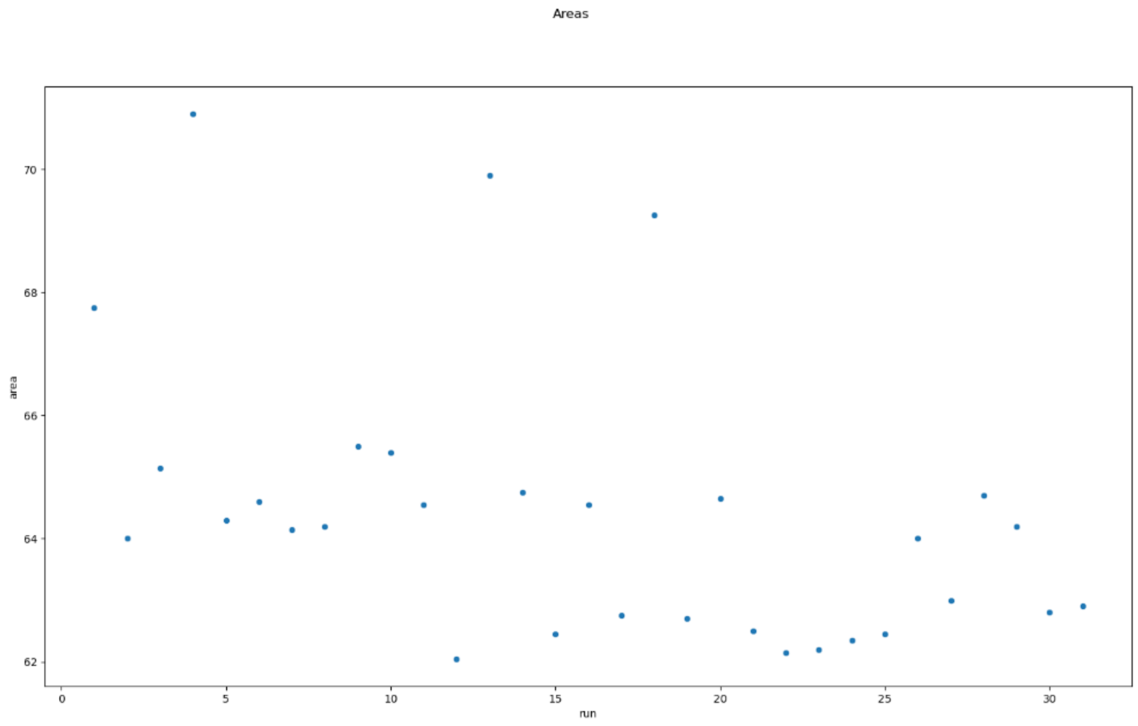


Figure A57. Scatter plot of the areas for Koa on the client side using the employees payload.

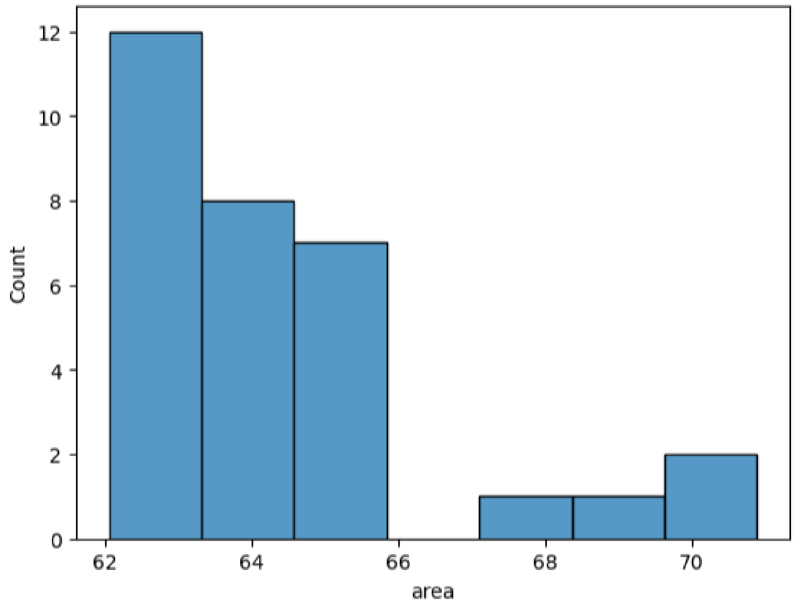


Figure A58. Histogram plot of the areas for Koa on the client side using the employees payload.

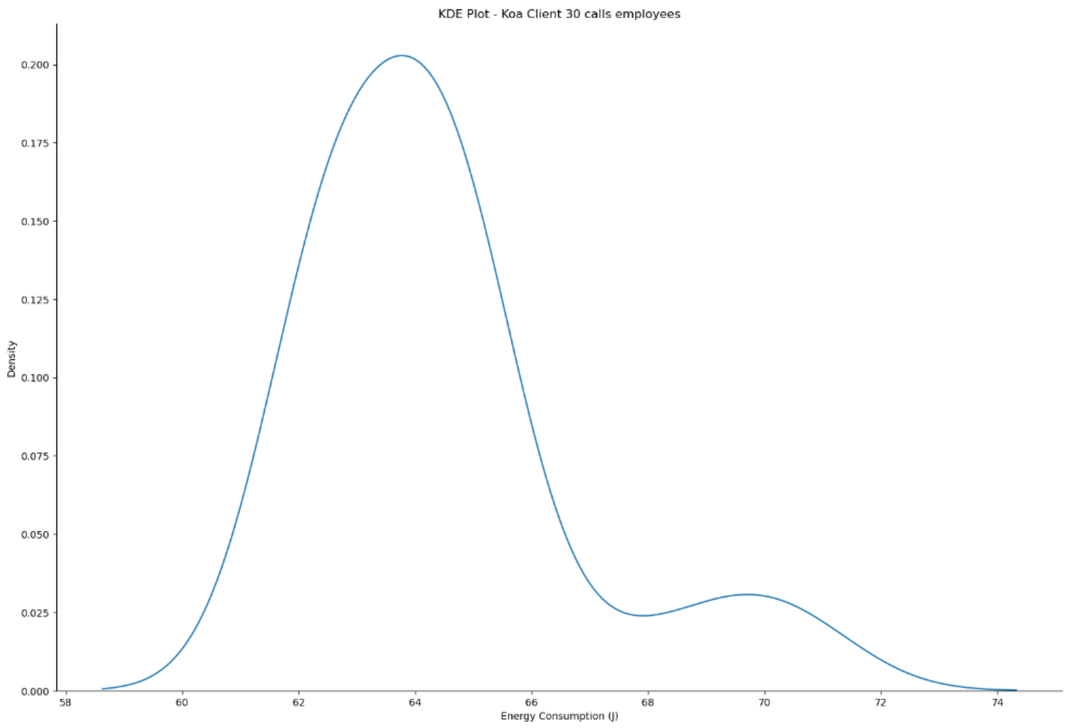


Figure A59. KDE plot of the areas for
Koa on the client side using the employees payload.

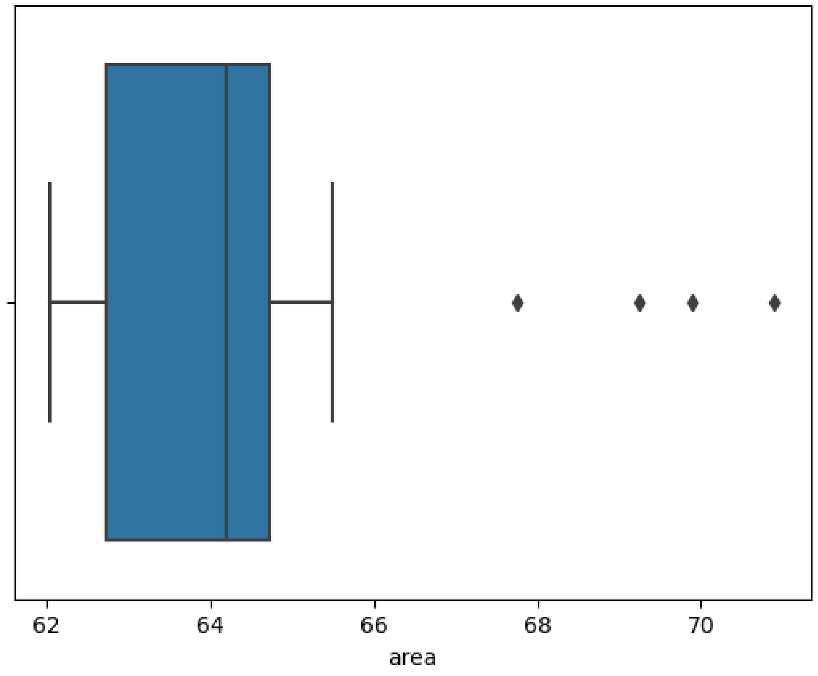


Figure A60. Box plot of the areas for
Koa on the client side using the employees payload.

9.4.15. Koa Client Departments

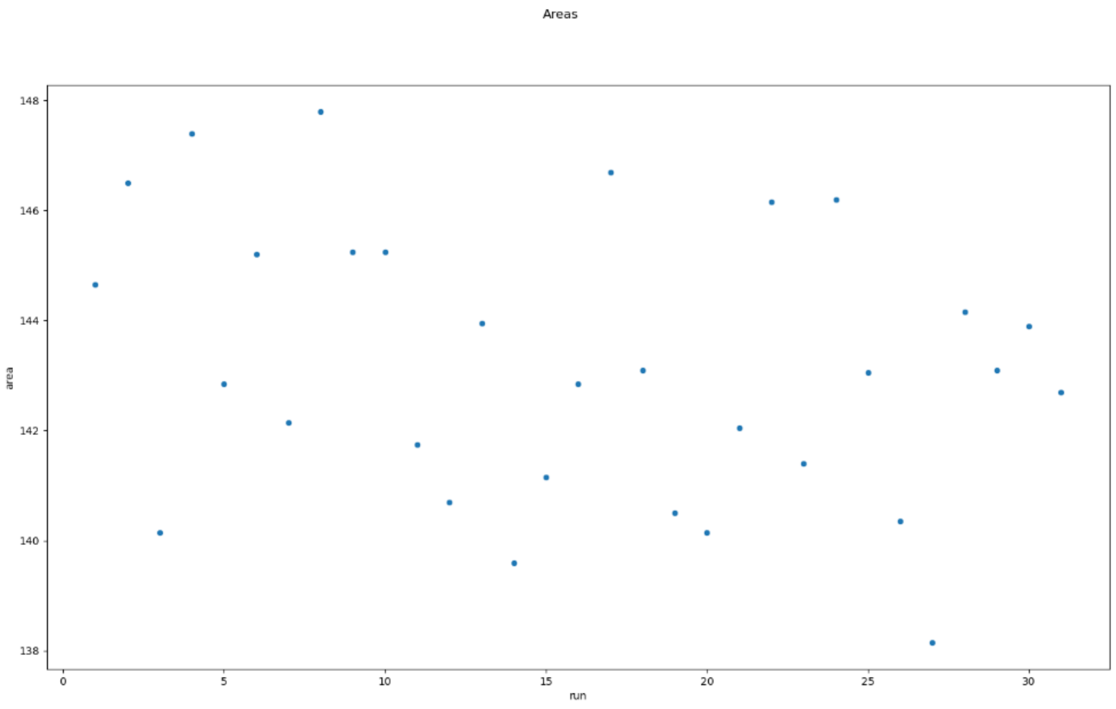


Figure A61. Scatter plot of the areas for Koa on the client side using the departments payload.

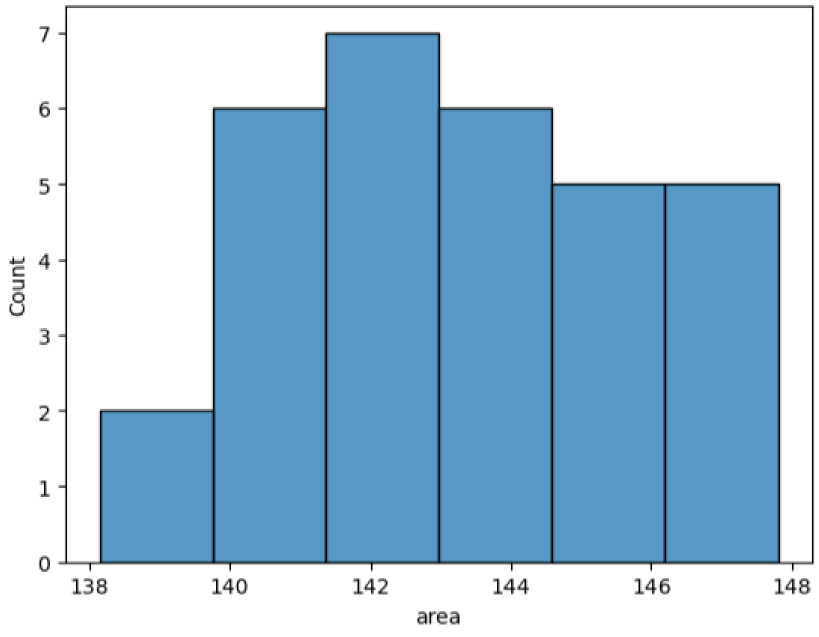


Figure A62. Histogram plot of the areas for Koa on the client side using the departments payload.

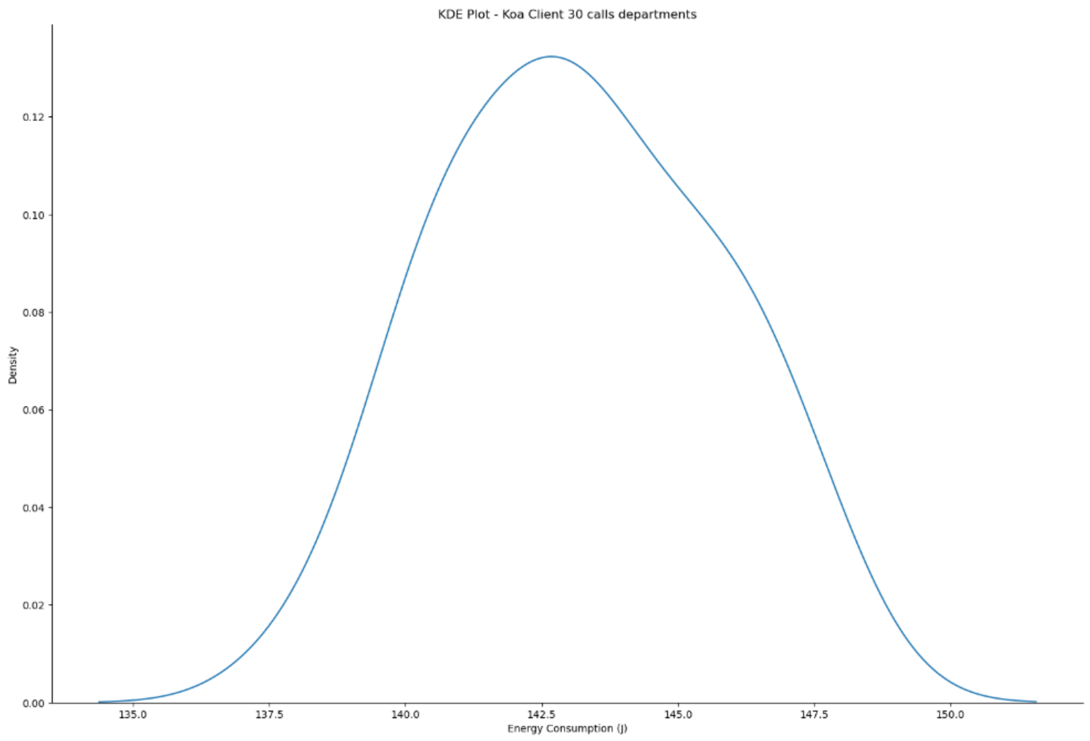


Figure A63. KDE plot of the areas for Koa on the client side using the departments payload.

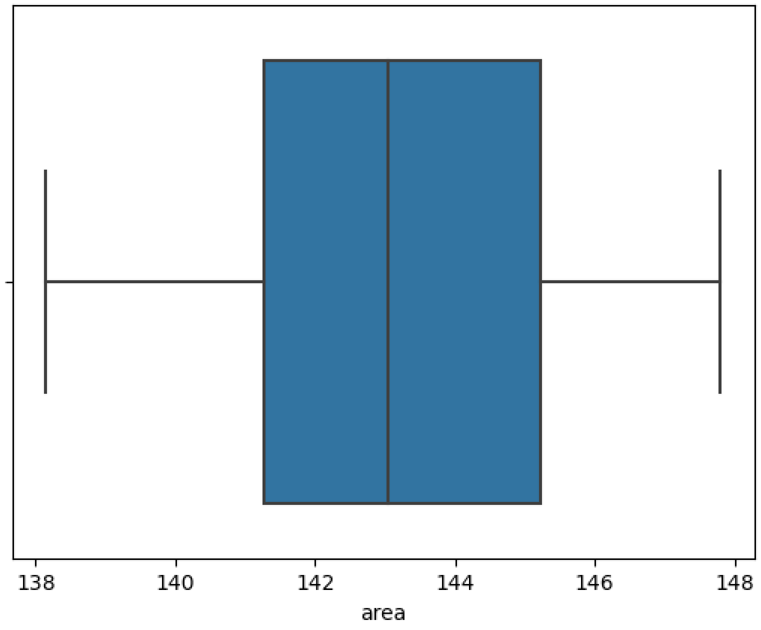


Figure A64. Box plot of the areas for Koa on the client side using the departments payload.

9.4.16. A/B test between Express Server Employees and Nest Server Employees

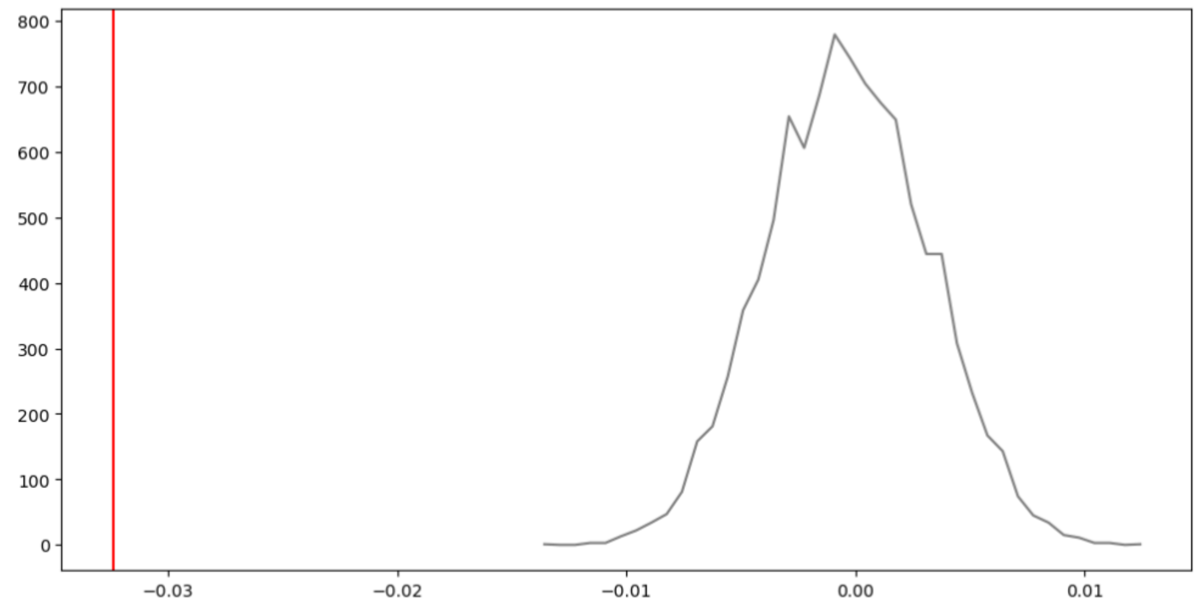


Figure A65. Histogram of the mean differences between Express and Nest, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.17. A/B test between Express Server Employees and Fastify Server Employees

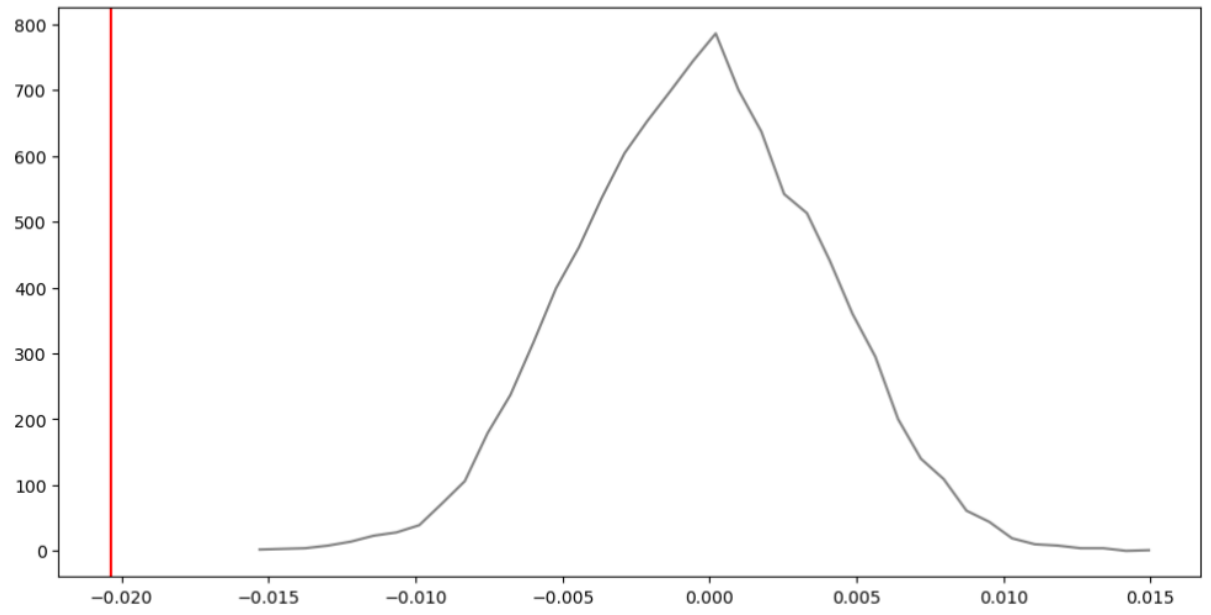


Figure A66. Histogram of the mean differences between Express and Fastify, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.18. A/B test between Express Server Employees and Koa Server Employees

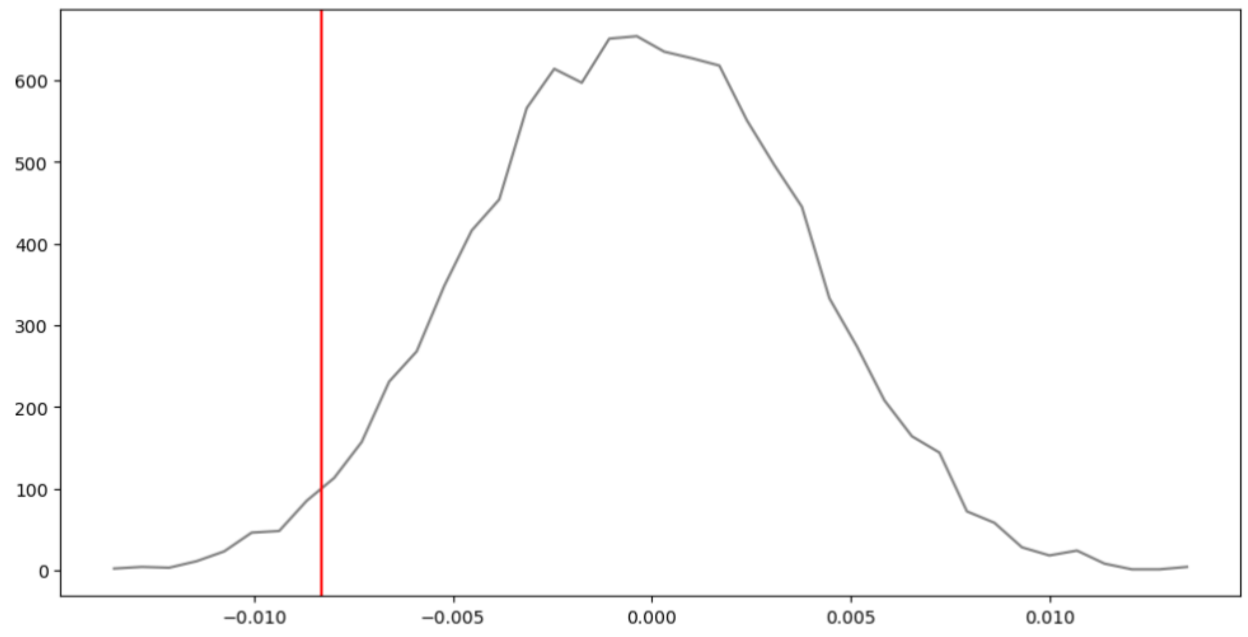


Figure A67. Histogram of the mean differences between Express and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.19. A/B test between Nest Server Employees and Fastify Server Employees

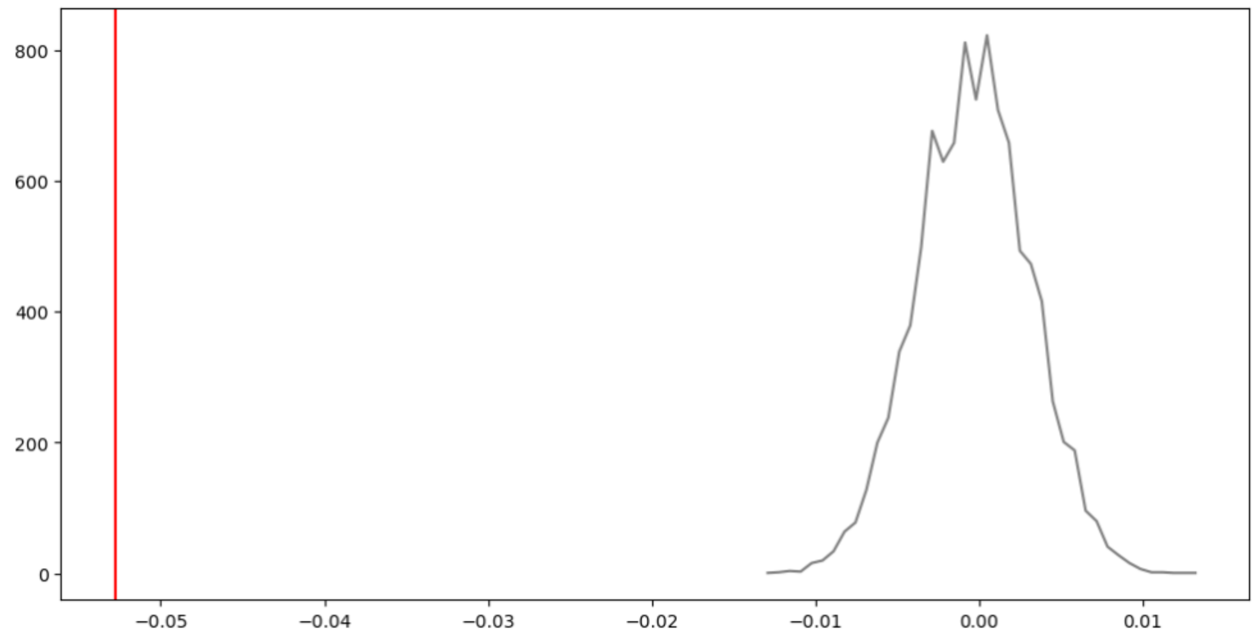


Figure A68. Histogram of the mean differences between Nest and Fastify, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.20. A/B test between Nest Server Employees and Koa Server Employees

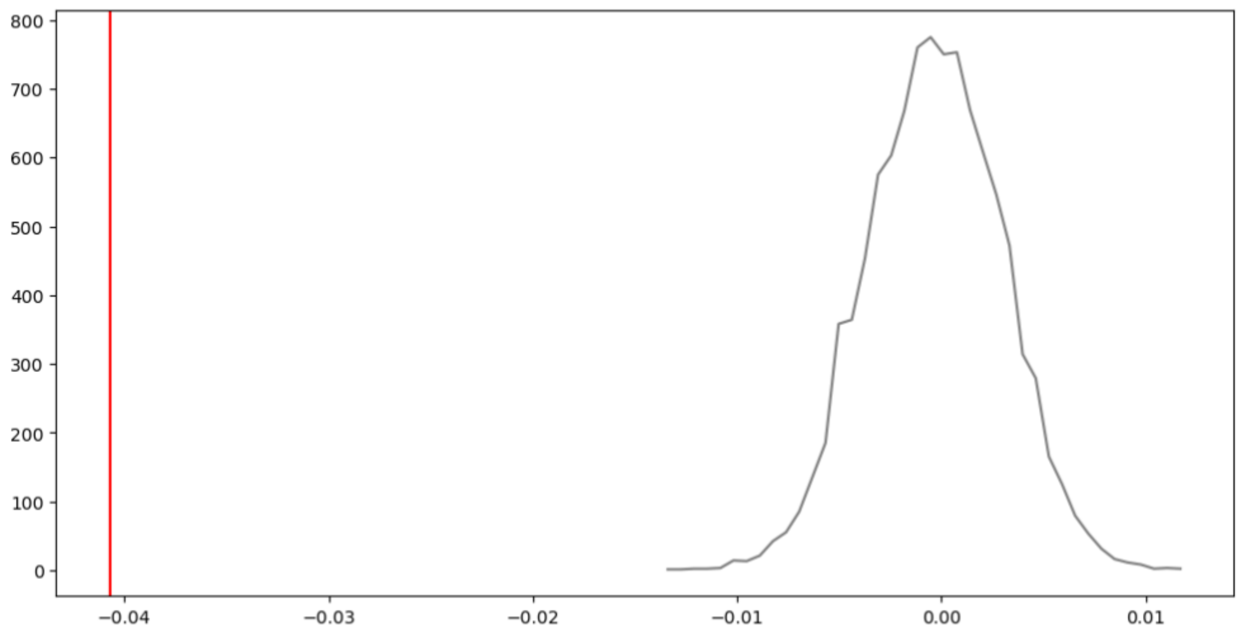


Figure A69. Histogram of the mean differences between Nest and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.21. A/B test between Fastify Server Employees and Koa Server Employees

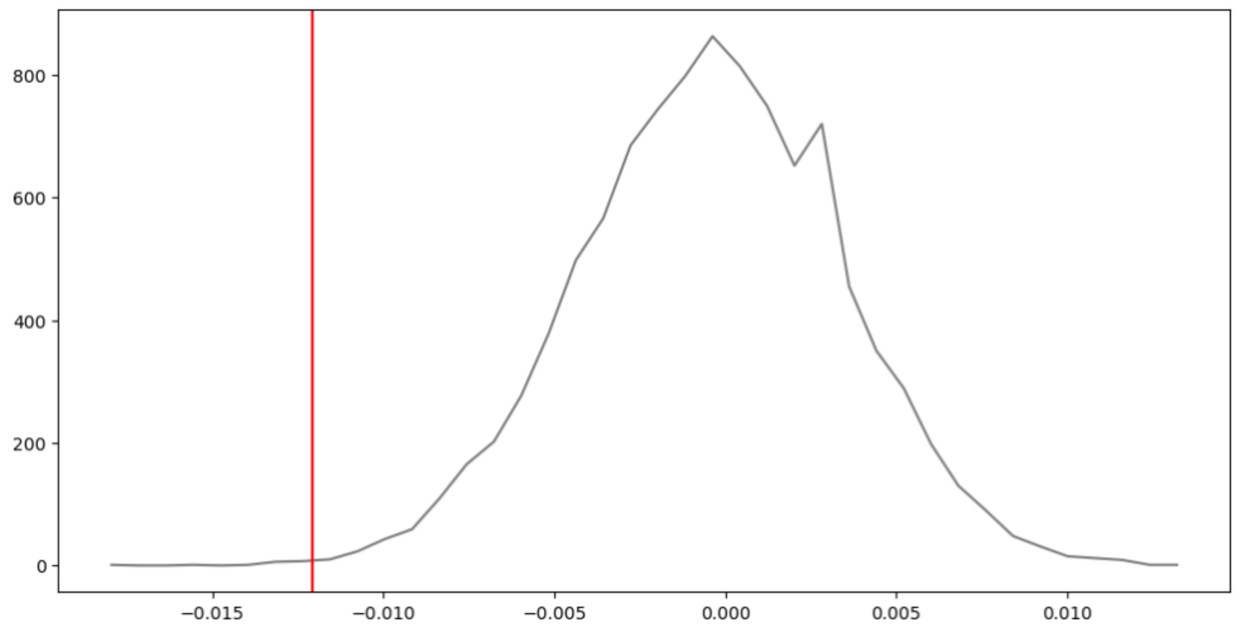


Figure A70. Histogram of the mean differences between Fastify and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.22. A/B test between Express Server Departments and Nest Server Departments

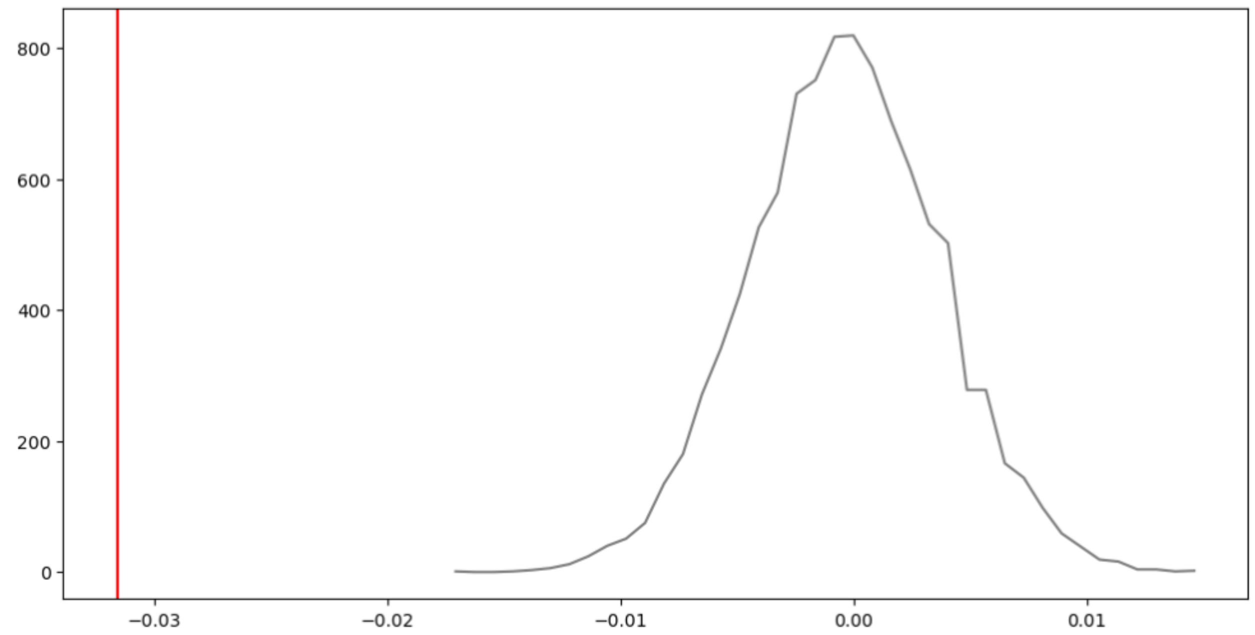


Figure A71. Histogram of the mean differences between Express and Nest, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.23. A/B test between Express Server Departments and Fastify Server Departments

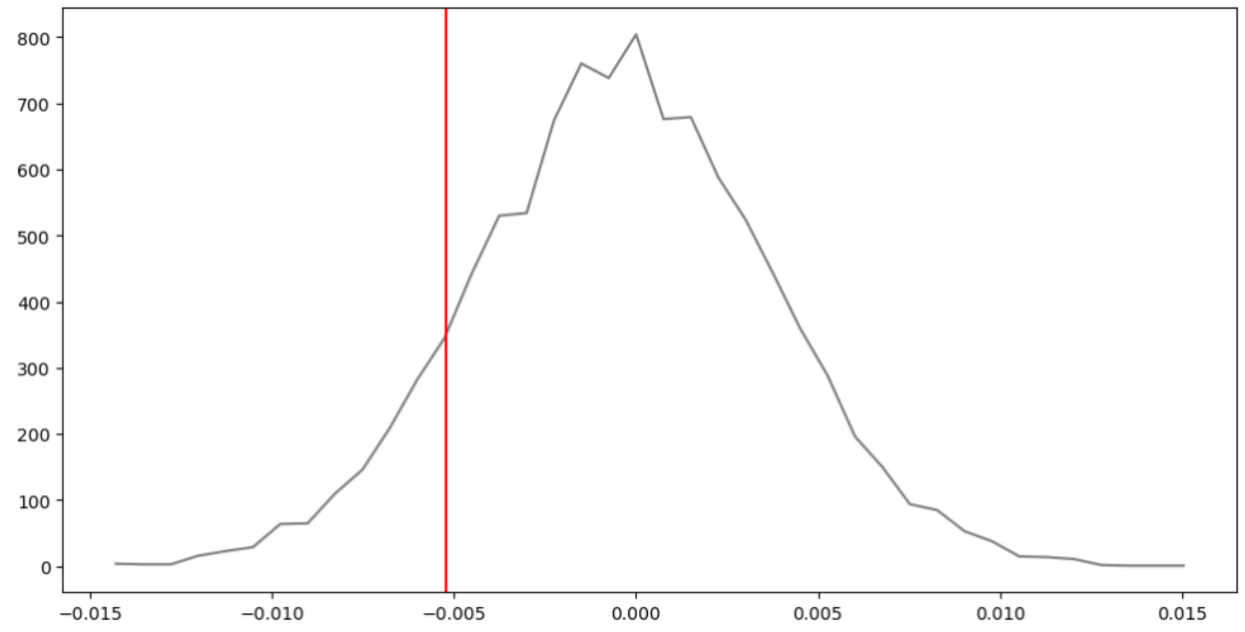


Figure A72. Histogram of the mean differences between Express and Fastify, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.24. A/B test between Express Server Departments and Koa Server Departments

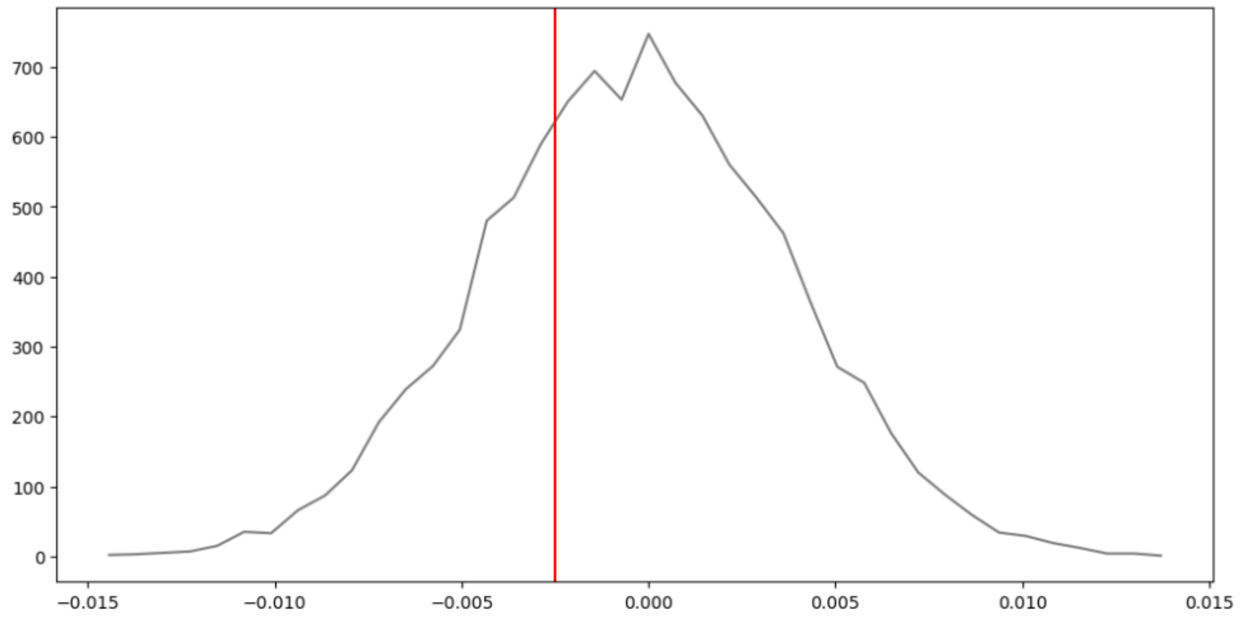


Figure A73. Histogram of the mean differences between Express and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.25. A/B test between Nest Server Departments and Fastify Server Departments

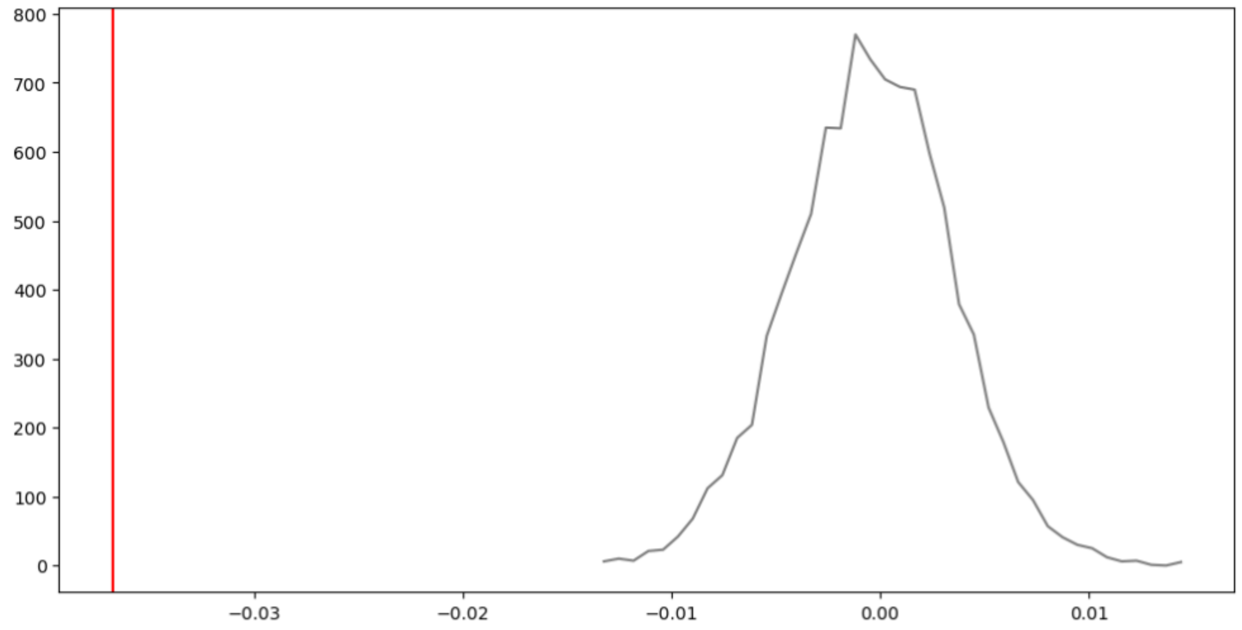


Figure A74. Histogram of the mean differences between Nest and Fastify, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.26. A/B test between Nest Server Departments and Koa Server Departments

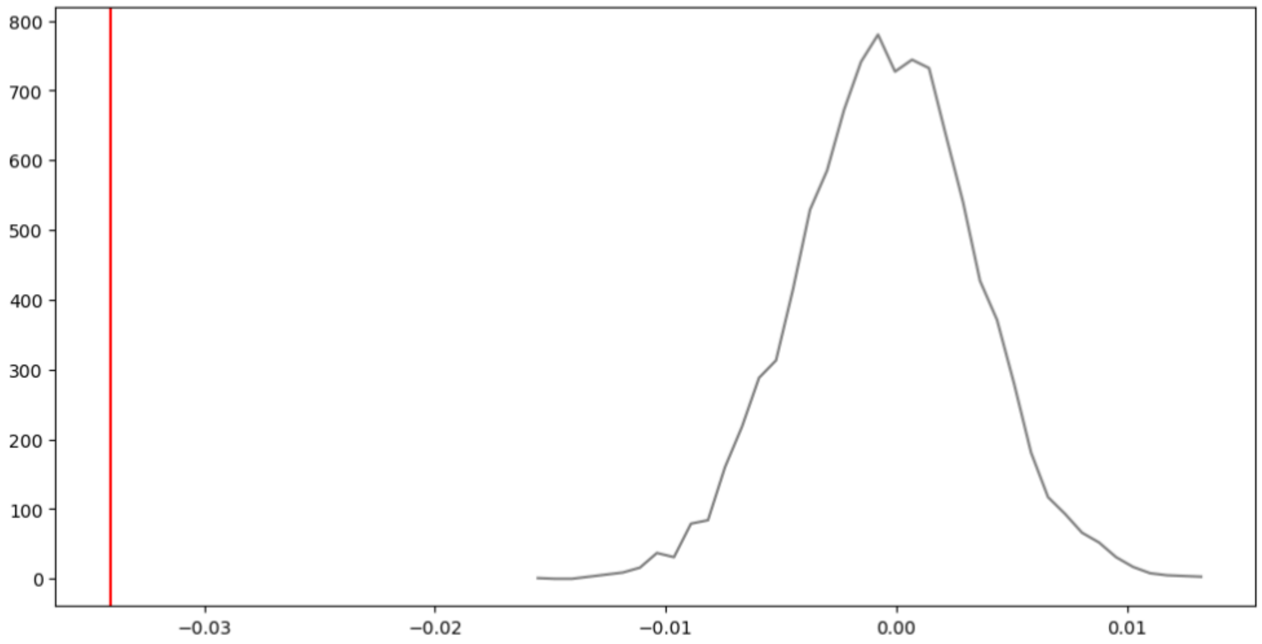


Figure A75. Histogram of the mean differences between Nest and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.27. A/B test between Fastify Server Departments and Koa Server Departments

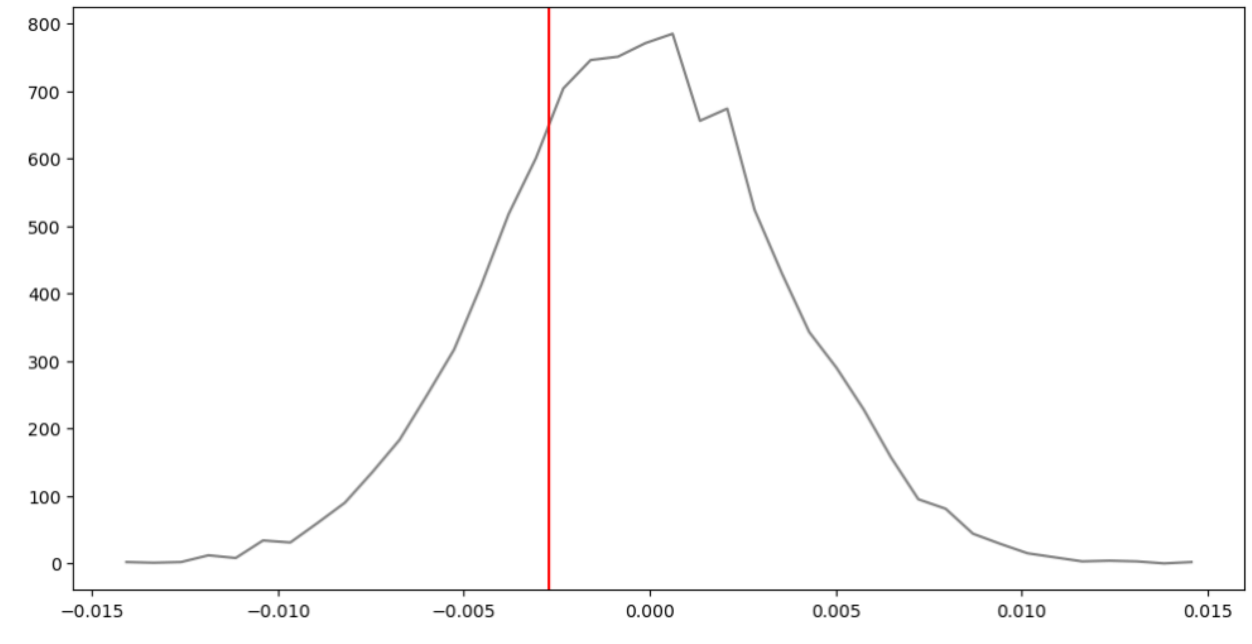


Figure A76. Histogram of the mean differences between Fastify and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.28. A/B test between Express Client Employees and Nest Client Employees

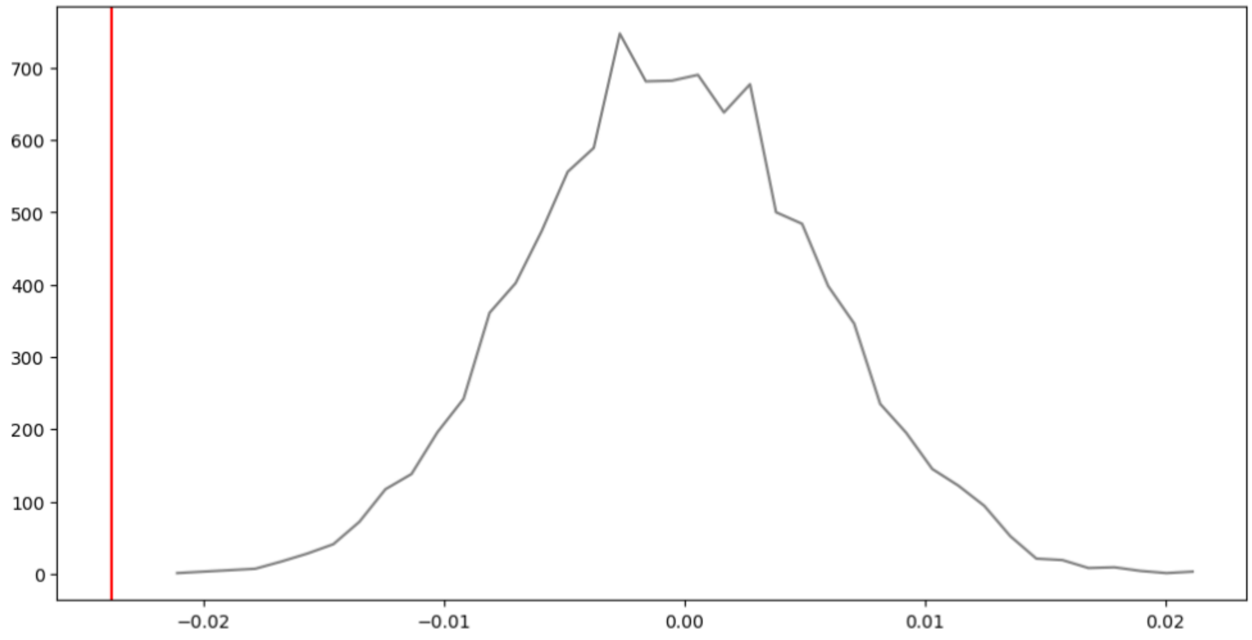


Figure A77. Histogram of the mean differences between Express and Nest, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.29. A/B test between Express Client Employees and Fastify Client Employees

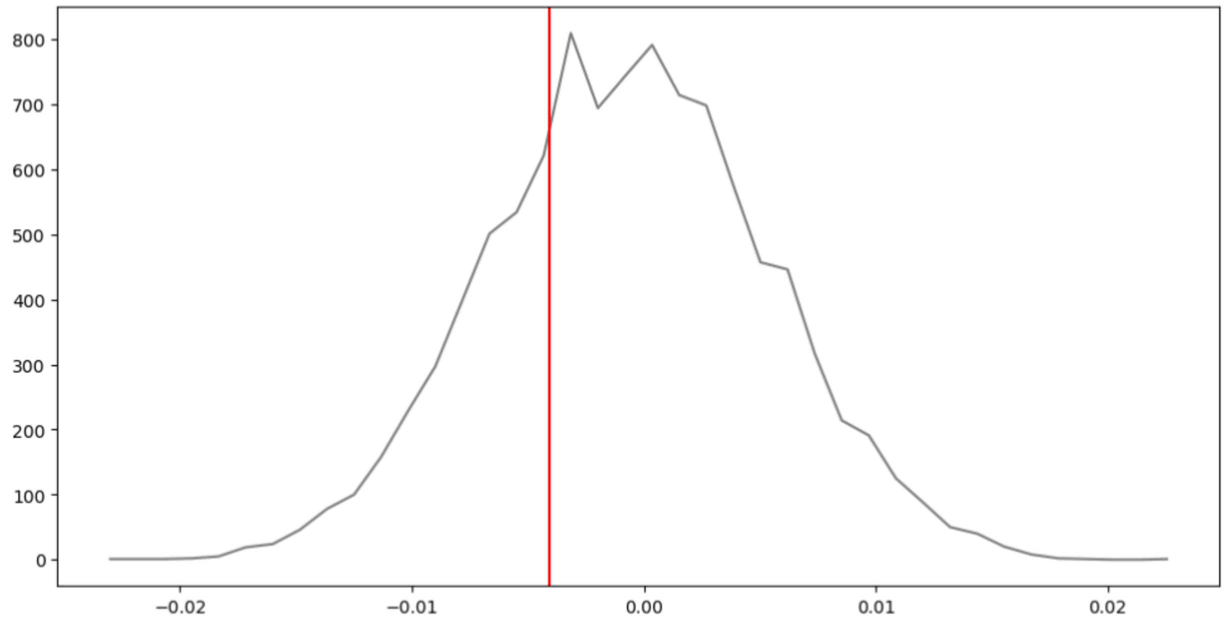


Figure A78. Histogram of the mean differences between Express and Fastify, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.30. A/B test between Express Client Employees and Koa Client Employees

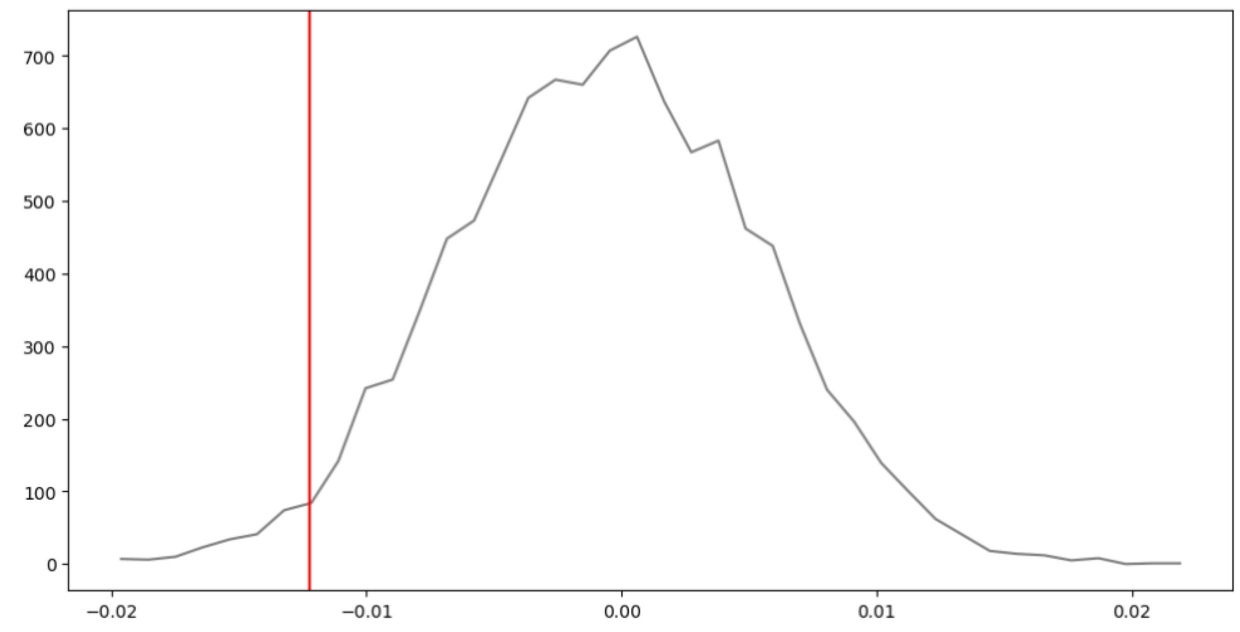


Figure A79. Histogram of the mean differences between Express and Nest, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.31. A/B test between Nest Client Employees and Fastify Client Employees

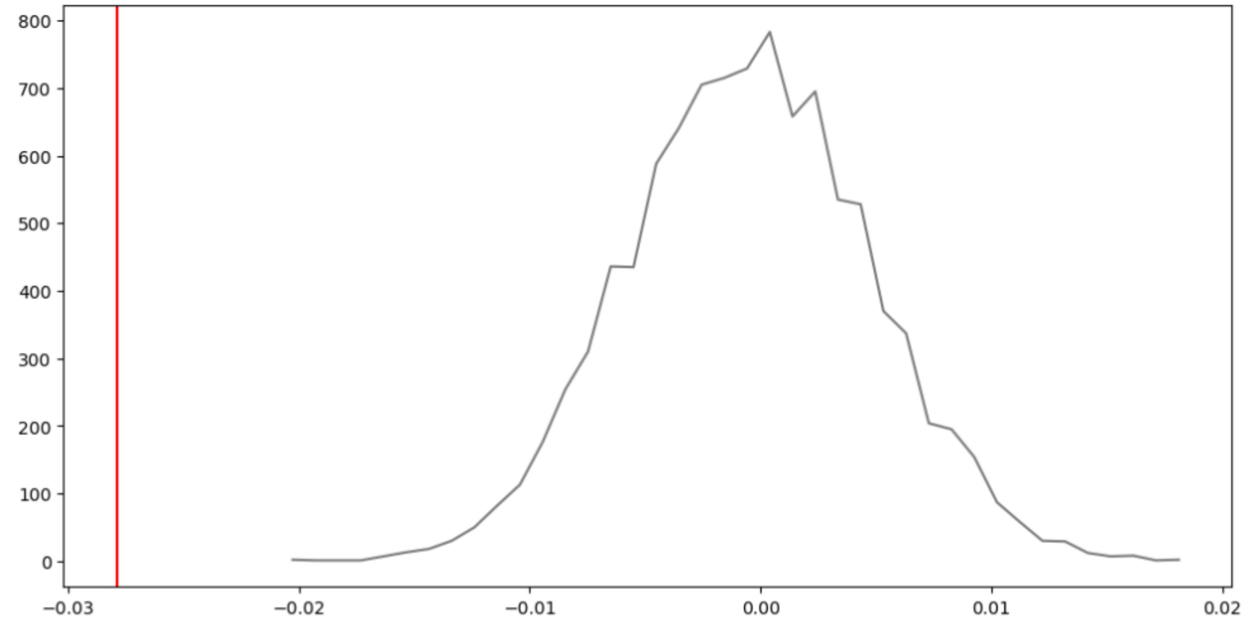


Figure A80. Histogram of the mean differences between Express and Fastify, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.32. A/B test between Nest Client Employees and Koa Client Employees

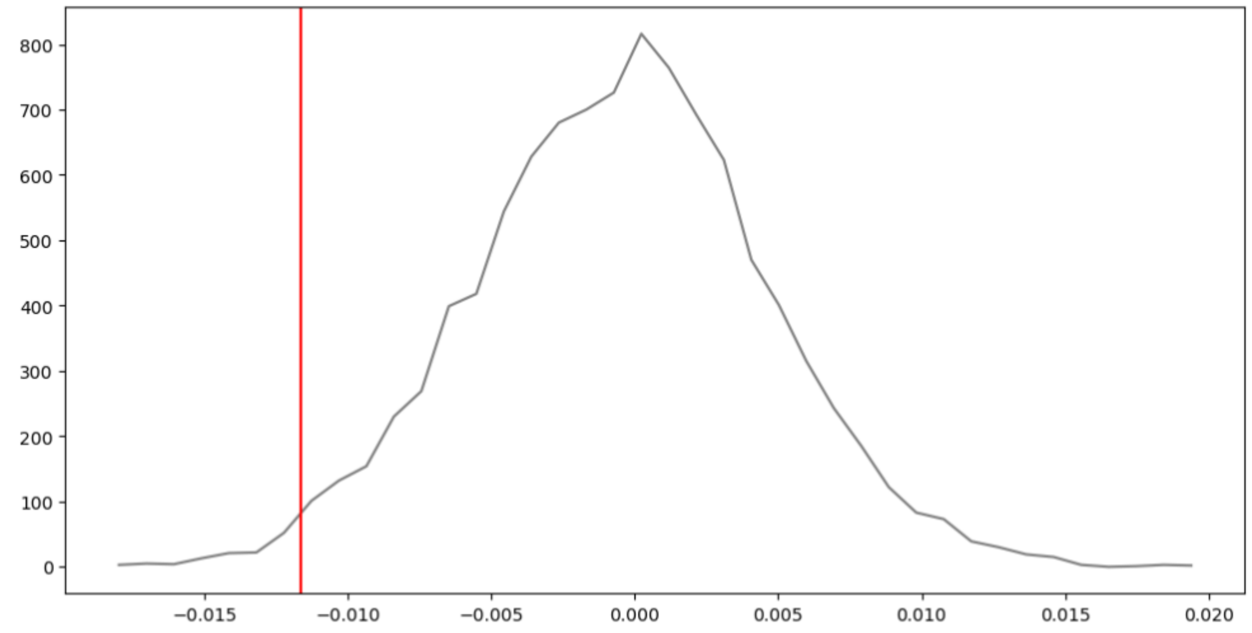


Figure A81. Histogram of the mean differences between Nest and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.33. A/B test between Nest Client Employees and Koa Client Employees

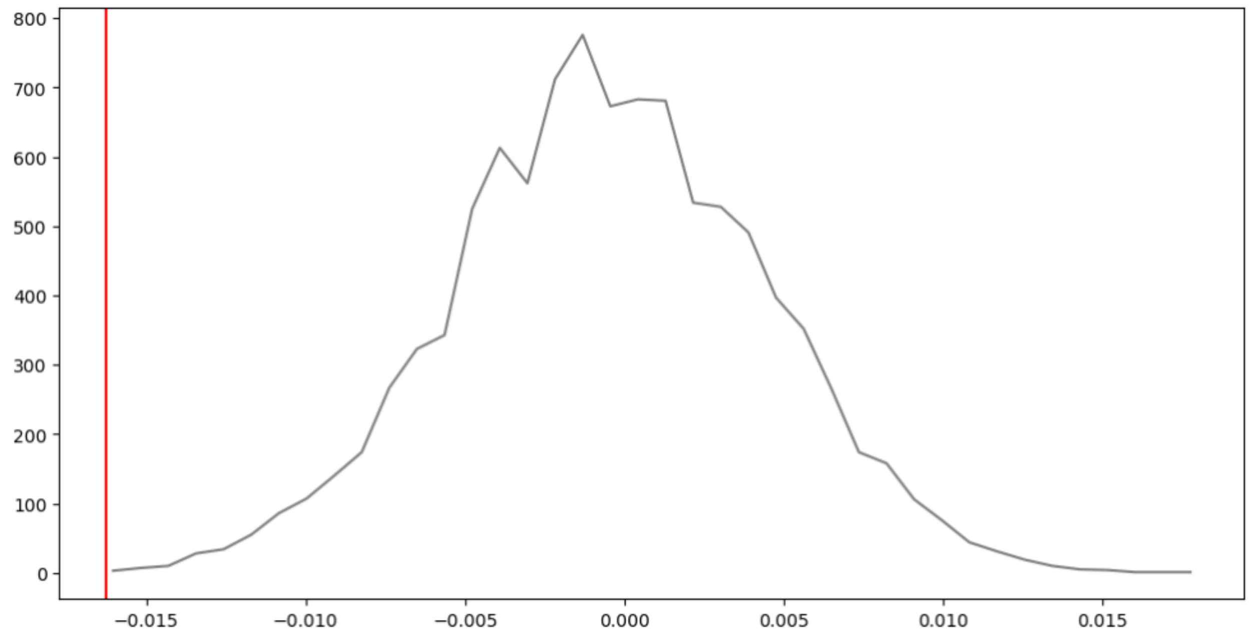


Figure A82. Histogram of the mean differences between Nest and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.34. A/B test between Express Client Departments and Nest Client Departments

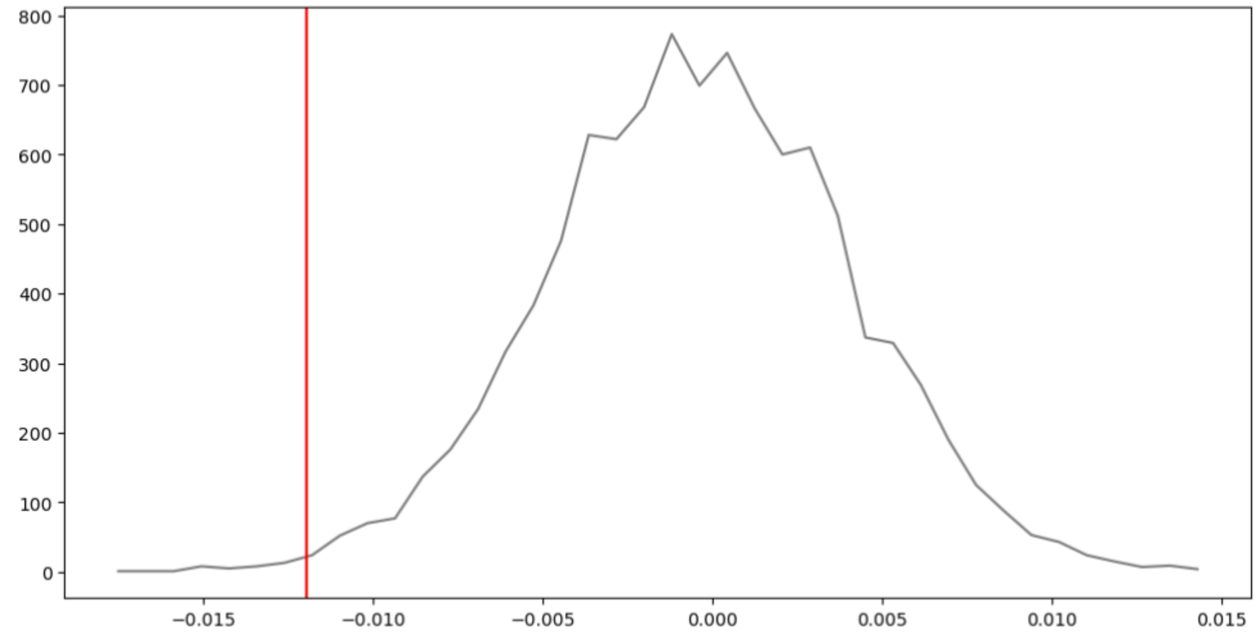


Figure A83. Histogram of the mean differences between Express and Nest, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.35. A/B test between Express Client Departments and Fastify Client Departments

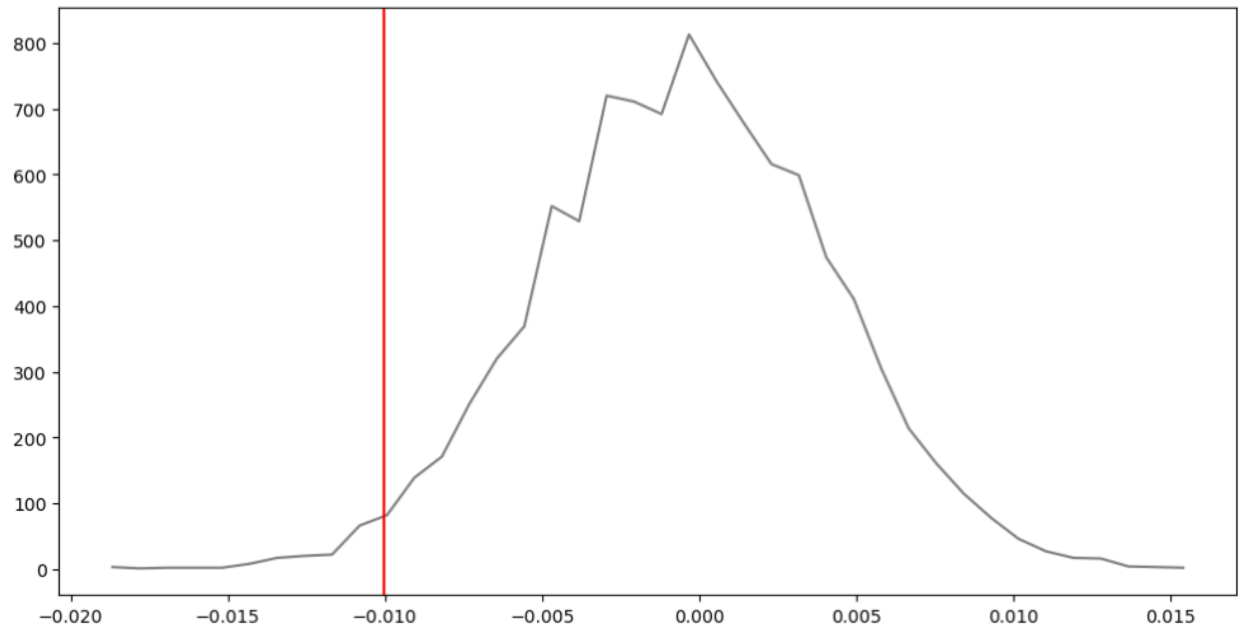


Figure A84. Histogram of the mean differences between Express and Fastify, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.36. A/B test between Express Client Departments and Koa Client Departments

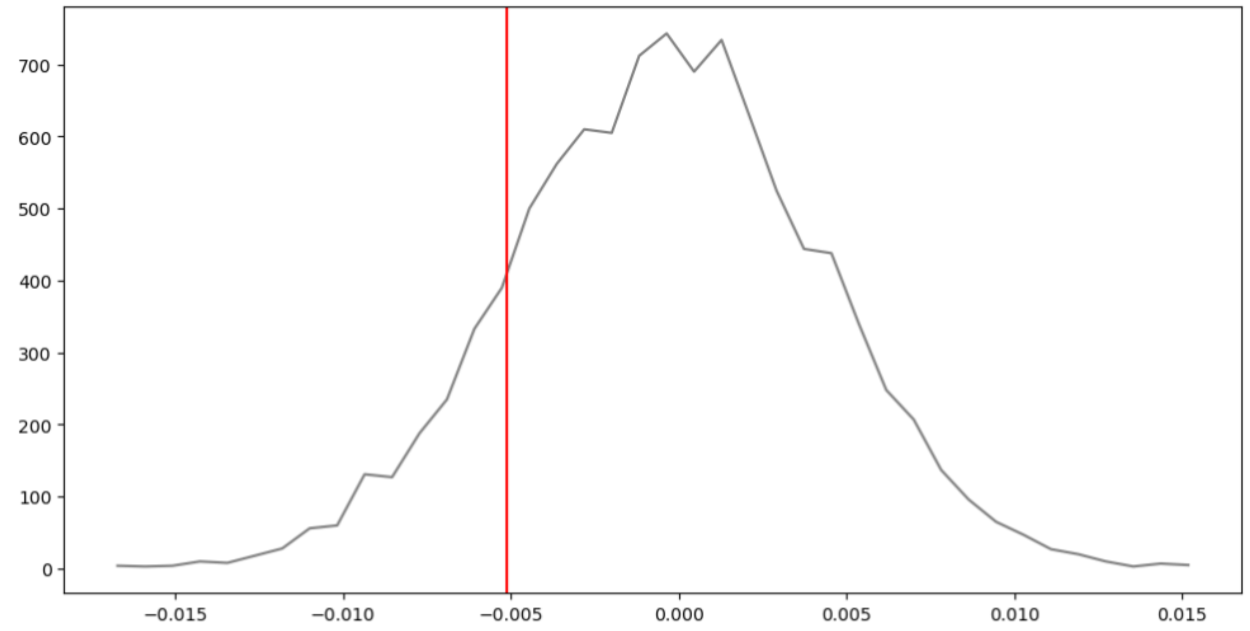


Figure A85. Histogram of the mean differences between Express and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.37. A/B test between Nest Client Departments and Fastify Client Departments

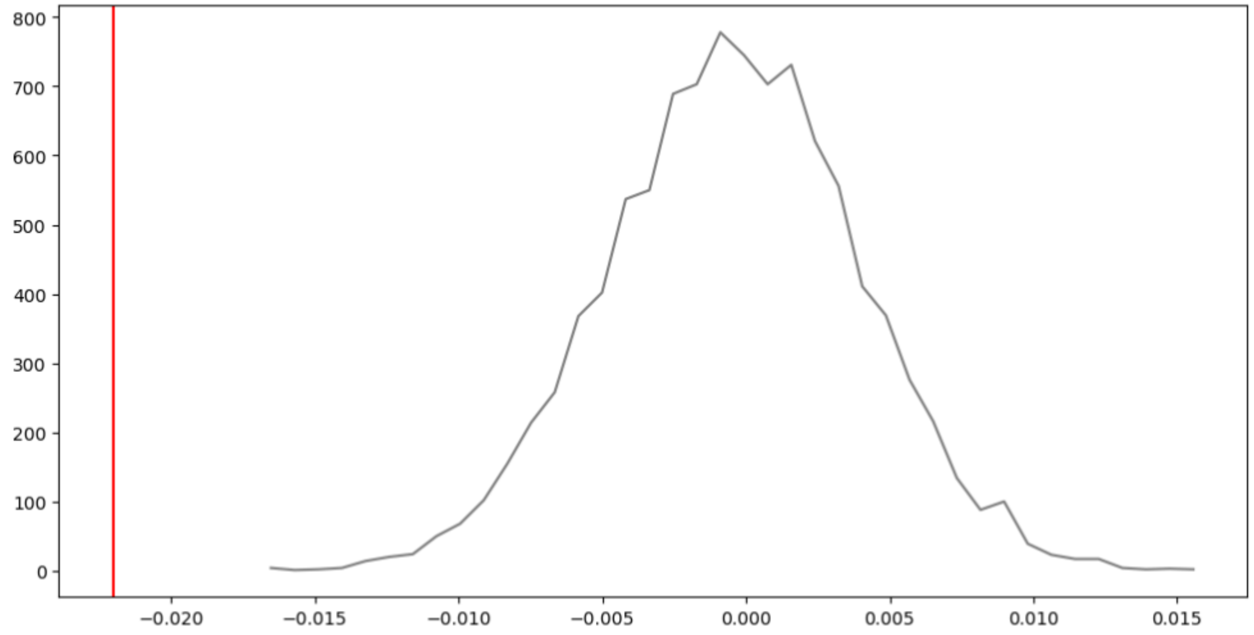


Figure A86. Histogram of the mean differences between Nest and Fastify, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.38. A/B test between Nest Client Departments and Koa Client Departments

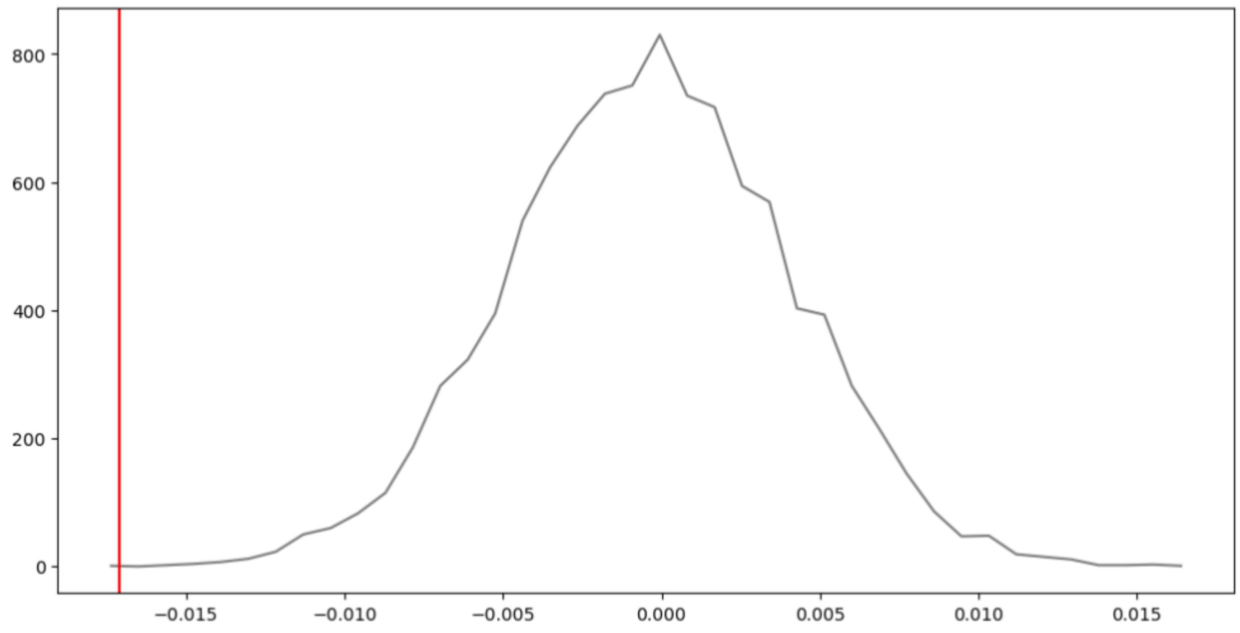


Figure A87. Histogram of the mean differences between Nest and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

9.4.39. A/B test between Fastify Client Departments and Koa Client Departments

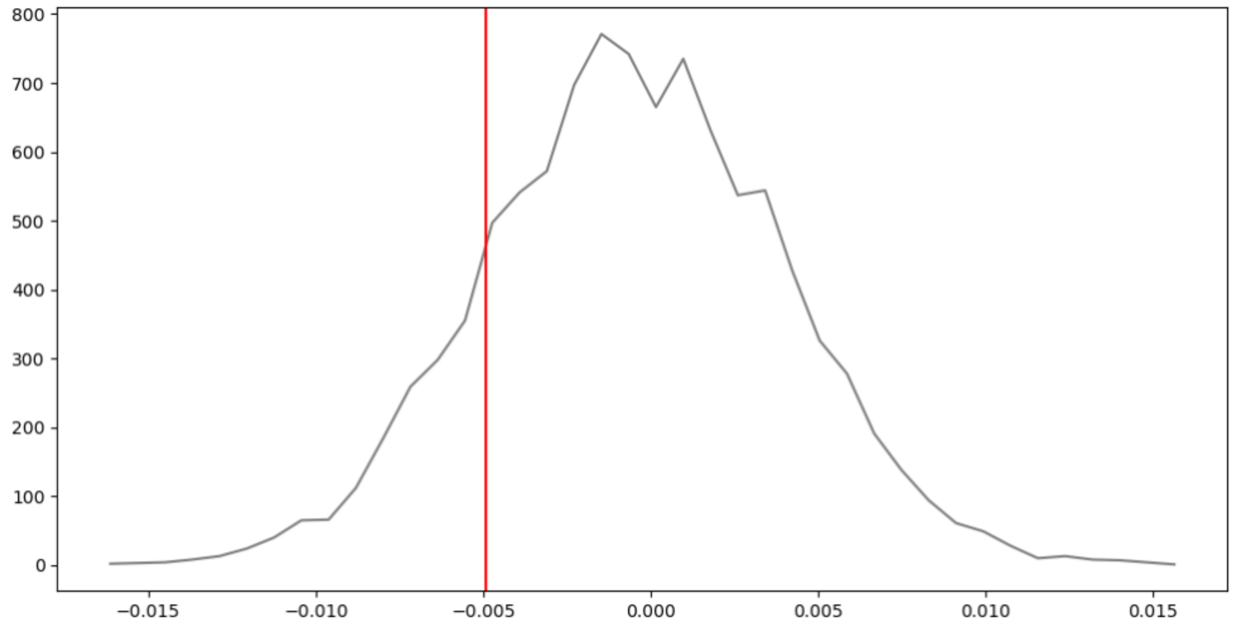


Figure A88. Histogram of the mean differences between Fastify and Koa, with distribution centered around 0. The red line is the difference between the averages of the original samples.

10.4. Bash script

```
#!/bin/bash

# only real time
TIMEFORMAT=%R
LOGFILE="results/runtime.log"

run_files() {

    for file in $1
    do
        for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30
        do
            echo $i"[$(date)] - before" >> $LOGFILE
            for j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30
            do
                while IFS= read -r line; do
                        echo "\nCurl command: curl" $line
                        curl $line
                done < $file
            done
            echo "\n--------------------------------: i:"$i
            echo $i"[$(date)] - after" >> $LOGFILE
            sleep 60
        done
        sleep 300
    done
}

rm -f $LOGFILE

pwd >> $LOGFILE
echo "[$(date)]" >> $LOGFILE

# Express
```

```
sleep 300
run_files "executables/express/employees.txt"
run_files "executables/express/departments.txt"
# Nest
run_files "executables/nest/employees.txt"
run_files "executables/nest/departments.txt"
# Fastify
run_files "executables/fastify/employees.txt"
run_files "executables/fastify/departments.txt"
# Koa
run_files "executables/koa/employees.txt"
run_files "executables/koa/departments.txt"

echo "done"
cat $LOGFILE
```