UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Artur Kasenõmm

Energy Matters: Evaluating JavaScript Asynchronous Patterns for Green Development

Bachelor's Thesis (9 ECTS)

Supervisor:
Hina Anwar, PhD

Tartu 2025

Energy Matters: Evaluating JavaScript Asynchronous Patterns for Green Development

Abstract:

The increasing energy consumption of the ICT sector presents environmental and economic concerns. JavaScript, the leading language for web development, is not the most energy-efficient compared to other programming languages. While optimization often focuses on technology choices or algorithmic improvements, the energy impact of different programming patterns for the same functionality in a particular language is understudied. Using Node.js, Perf, and controlled test cases simulating HTTP requests and file writing, this thesis compares the energy consumption of JavaScript's asynchronous patterns: callbacks, promises, and async/await. Statistical analysis revealed that the callback pattern consumed significantly less energy (around 6-7%) than the async/await pattern for HTTP requests with simulated latency. No significant energy difference was found for file writing. These findings suggest minor efficiency benefits for the callback pattern in specific scenarios. However, the considerable readability, maintainability, and error handling advantages of the promise and the async/await patterns likely make them preferable in most development contexts, highlighting the trade-off between minor energy gains and developer productivity.

Keywords:

JavaScript, Energy Efficiency, Asynchronous Programming, Green Software Development, Web Development

CERCS: P170 Computer science, numerical analysis, systems, control

JavaScripti Asünkroonsete Mustrite Energiatõhususe Analüüs

Lühikokkuvõte:

IKT-sektori suurenev energiatarbimine koormab keskkonda ja majandust. Veebiarenduses domineeriv JavaScript ei kuulu teiste programmeerimiskeeltega võrreldes kõige energiasäästlikumate hulka. Ehkki tarkvara optimeerimisel pööratakse enamasti tähelepanu tehnoloogia valikule või algoritmide täiustamisele, on ühe ja sama funktsionaalsuse saavutamiseks kasutatavate programmeerimismustrite energiamõju ühe keele piires seni vähe käsitlust leidnud. Käesolevas lõputöös võrreldakse JavaScripti asünkroonsete mustrite *callback*, *promise* ning *async/await* energiakulu. Uurimus põhineb Node.js käituskeskkonnas ja Perfi mõõtmistööriistaga tehtud katsetel milles jäljendati HTTP-päringuid ja faili kirjutamist. Statistilise andmeanalüüsi käigus leidsime, et fikseeritud viitajaga HTTP-päringute korral tarbis *callback* muster vähem energiat (ligikaudu 6–7%) kui *async/await* lahendus. Failidesse kirjutamise puhul aga mustrite vahel märkimisväärset erinevust energiatarbimises ei täheldatud. Need tulemused viitavad sellele, et *callback* muster võib teatud olukordades olla energiatõhusam valik. Sellegipoolest tuleb arvestada *promise* ja *async/await* mustrite eelistega koodi loetavuses, hooldatavuses ning veakäsitluses. Enamasti kaaluvad need eelised üles väikese energiasäästu, mistõttu on *promise* ja *async/await* mustrid paljudes olukordades tõenäoliselt parem valik. Lõplik otsus sõltub siiski konkreetsest olukorrast ning sobiva tasakaalu leidmisest mõningase energiasäästu ja arendaja töö mugavuse ning tõhususe vahel.

Võtmesõnad:

JavaScript, Energiatõhusus, Asünkroonne Programmeerimine, Roheline Tarkvaraarendus, Veebiarendus

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

1 Introduction

The information and communication technology (ICT) sector is growing rapidly and is becoming more integrated into modern life. This growth has led to an increase in global energy consumption. Estimates indicate that the ICT sector consumed 7% of global energy in 2020, with projections for 9% by 2025 and 14% by 2030 [1]. In terms of electricity, the sector is responsible for approximately 4% of global electricity consumption [2], with data centers and data transmission networks accounting for approximately 1-1.5% [3].

The increasing energy demand in the ICT sector creates environmental challenges through its carbon footprint, estimated at 2.1-3.9% of global emissions [4], a figure that will likely increase with the continued growth of digital technologies. Furthermore, even seemingly minor software inefficiencies can accumulate into substantial energy waste when multiplied by billions of users and devices. Addressing these challenges aligns with global sustainability goals, like the United Nations Sustainable Development Goal 7, which targets doubling the global rate of improvement in energy efficiency by 2030 [5].

1.1 Energy Efficiency in ICT

Efficiency and performance challenges have historically been addressed by upgrading hardware and adding more resources. However, the effectiveness of hardware upgrades is diminishing due to the deceleration of Moore's law[1] [6, 7]. Similarly, scaling through additional hardware faces physical limitations, particularly in resource-constrained devices like mobile phones, which have space, computing power, and battery capacity constraints. Inefficient software impacts user experience on such devices through faster battery depletion and reduced performance [8]. The issue is significant: mobile devices accounted for over half of all web traffic in 2022 [9]. Hardware constraints suggest software optimization as a method to improve energy efficiency.

Software optimization can be applied at multiple levels, including architectural design, technology selection, and algorithm implementation. While it is clear that poor implementation of an algorithm can significantly affect software energy efficiency, technology

[1]Moore's law, proposed by Gordon Moore in 1965, predicted that the number of transistors on a microchip would double every two years.

choices, such as language selection, also influence software energy consumption.

Studies show systems languages like C are generally more energy-efficient than higher-level languages like JavaScript [10]. While selecting the most energy-efficient programming language might seem ideal from an environmental perspective, factors such as developer productivity, language support, and ecosystem influence technology choices.

1.2 JavaScript in Web Development

An example of language selection trade-offs can be seen in web development, where JavaScript is the dominant programming language despite its relative energy inefficiency. JavaScript has been the most popular programming language for 11 consecutive years, according to Stack Overflow's Developer Survey [11]. GitHub's 2024 "State of the Octoverse" report [12] identified JavaScript as the most used language in its repositories in the preceding years. GitHub hosts over 420 million repositories [13]. The language's reach extends even further, with W3Tech reporting that 98.9% of surveyed websites implement JavaScript for client-side programming [14].

JavaScript's dominance extends to platforms serving billions of users. According to Singh [15], many technology companies such as Alphabet, Meta Platforms, and Uber Technologies use JavaScript frameworks to power their web applications (like Google[2], Facebook[3], Instagram[4], and Uber[5]). Meta Platforms alone serves a large user base, with 3.43 billion daily active users across its applications [16].

Despite its prevalence, JavaScript is not quite energy-efficient. Comparative studies show that it consumes approximately 4.45 times more energy than C and ranks 17th out of 27 popular programming languages in efficiency [10]. This can be attributed to features like garbage collection, just-in-time compilation, and dynamic typing. Although these features support developer productivity and code flexibility, they contribute to more runtime computation and increased energy consumption. Optimizing implementation within the selected language represents another approach to improving energy efficiency.

We are presented with an intriguing research opportunity. Given JavaScript's likely

[2]https://www.google.com/
[3]https://www.facebook.com/
[4]https://www.instagram.com/
[5]https://www.uber.com/

continued dominance in web development due to its ecosystem and developer familiarity, how can we optimize energy consumption within the constraints of this language? Instead of suggesting a complete language replacement, which is often unrealistic, writing energy-efficient JavaScript code could be a way to reduce energy consumption.

Although many studies [17, 18, 19] in software energy efficiency have focused on comparing different languages and code implementations, less attention has been paid to how different ways of expressing the same functionality within a particular language could affect energy consumption. Whether choices between coding approaches with different syntax and semantics but similar functionality result in significant differences in energy consumption is an area open for further research. Investigating this requires identifying a domain in JavaScript where developers have multiple coding approaches to implement some functionality.

1.3 Asynchronous Patterns

Asynchronous programming allows operations to run simultaneously without blocking the main program thread, which enables tasks with potential latency to proceed concurrently while other code executes [20]. Asynchronous programming is essential in modern JavaScript development, particularly for creating responsive web applications. High-traffic applications from the companies mentioned earlier (e.g., Facebook, Instagram) rely heavily on asynchronous operations for common use cases like fetching data from APIs, handling user interactions, and updating interfaces dynamically. Given the asynchronous nature of these web operations and the fact that they use JavaScript for web interfaces, these applications probably use the tools available in JavaScript to handle asynchronous tasks.

JavaScript offers three primary patterns for implementing asynchronous tasks: the callback pattern, the promise pattern, and the async/await pattern. In this paper, these specific structural and syntactical methods for handling asynchronous tasks are referred to as asynchronous patterns. Chapter 3.1 provides a detailed overview of these asynchronous patterns.

1.4 Problem Statement and Research Objectives

JavaScript's widespread use means that investigating energy-efficient asynchronous coding practices could reduce energy consumption in large applications, potentially reducing environmental impacts, improving device battery life, and reducing operational costs. One particularly important aspect of JavaScript development is asynchronous programming. JavaScript provides three patterns to handle asynchronous operations: the callback pattern, the promise pattern, and the async/await pattern. These three asynchronous patterns achieve similar functionality but differ in syntax and semantics. Although energy consumption differences in syntax alone might be minor, semantic differences can create more variance as they can affect how CPU, memory, and other system resources are used. Given these differences, developers aiming to write asynchronous JavaScript code energy-efficiently face uncertainty when choosing among these patterns, as limited guidelines regarding their energy consumption are available.

Existing research [19, 21, 22] investigates JavaScript energy consumption but does not explicitly target our research direction. Similarly, studies focusing on asynchronous patterns in JavaScript are uncommon [23, 24]. This area is largely underexplored, leaving developers without information to select the most energy-efficient asynchronous coding approaches when energy efficiency is a priority.

This thesis measures and compares the energy consumption of JavaScript's various asynchronous patterns to determine whether choosing one asynchronous pattern over another for implementing the same functionality can lead to significant energy savings. The findings will contribute to research on energy-efficient web development by providing data to developers seeking improvements in energy efficiency.

1.5 Thesis Structure

The rest of the thesis is structured as follows. Chapter 2 reviews the existing literature on energy efficiency in software development and JavaScript asynchronous programming, identifying gaps in current research relating to asynchronous patterns. Chapter 3 details the research methodology, including experimental design, measurement tools, data collection, and data analysis. Chapter 4 presents the results, discusses the findings, and concludes with recommendations for energy-efficient asynchronous programming in JavaScript and suggestions for future research.

1.6 Use of AI

In this thesis, Microsoft Copilot (2025) [25] was used to edit and improve the formatting of existing text during the final stages of writing. It also assisted in creating some bash scripts for automating parts of the experiment, as further detailed in Chapter 3.

2 Literature Review

This chapter reviews the literature relevant to the topic of this thesis. It begins with research on how language and coding choices impact software energy consumption, and then narrows to the JavaScript landscape, covering optimizations and comparisons. The review then examines studies on asynchronous programming in JavaScript and its specific asynchronous patterns. Finally, it highlights the lack of studies on the energy efficiency of these asynchronous patterns, noting the need for a comparative study.

Previous research has established that the choice of programming language and specific coding decisions significantly influence energy consumption. At the language level, Pereira et al. [17, 10] have established comparative energy efficiency rankings across various programming languages. At the code level, Georgiou et al. [18] have demonstrated how different implementations affect computational workload and overall energy efficiency. Specifically within the JavaScript ecosystem, several studies have investigated optimization techniques: Selakovic and Pradel [19] explored general JavaScript optimizations, Kyriakou and Tselikas [26] compared JavaScript with WebAssembly on Android devices, and Patrou et al. [21] focused on resource utilization in Node.js's asynchronous I/O operations. While these studies provide a valuable foundation, they do not specifically examine the asynchronous patterns in JavaScript, which is the focus of this thesis.

Several studies [27, 28, 29] have focused on investigating asynchronous patterns in JavaScript beyond general performance enhancements. Research by Turcotte [27] examined asynchronous code refactoring techniques but did not address energy consumption aspects. Luong and Canh [28] analyzed the differences and benefits of modern asynchronous programming methods compared to traditional approaches. Fritz and Zhao [29] traced the evolution from the callback pattern to the promise pattern and even developed their own asynchronous syntax to improve the developer experience. Nevertheless, these studies have not precisely done what this research aims to accomplish.

Gokhale et al. [23] took a step closer to understanding the energy impact of asynchronous code, showing that migrating synchronous APIs to their asynchronous equivalents using the async/await pattern can significantly improve scalability and execution time. They developed a refactoring technique to perform such transformations. Of the 256 transformations tested, 244 resulted in reduced execution time without alter-

ing program behavior. These results highlight that asynchronous implementations can yield performance gains. However, they noted that for non-concurrent operations, asynchronous code does not guarantee better execution time due to the overhead of managing and creating asynchronous structures at the code level.

Efforts have also been made to identify and resolve issues within asynchronous code. Turcotte et al. [22] introduced DrAsync, a tool designed to visualize and pinpoint anti-patterns in asynchronous JavaScript, demonstrating that optimizing inefficient use of the promise pattern reduced energy consumption by approximately 4%. Similarly, Alimadadi et al. [24] documented anti-patterns in asynchronous JavaScript, suggesting that a better grasp of these problematic patterns could lead to more effective solutions. Although these efforts represent progress, they also underscore the need for more detailed guidance on developing asynchronous, energy-efficient JavaScript.

Beyond addressing specific issues in asynchronous code, researchers have also considered broader concerns related to energy efficiency. Pinto and Castor [8] observed that energy efficiency is increasingly being recognized as a crucial consideration for software developers. This acknowledgment has led to the creation of general guidelines for energy-efficient software development by Cruz and Abreu [30] and Ren et al. [31]. However, these broader frameworks do not specifically address asynchronous JavaScript code. Therefore, a comprehensive investigation of the energy efficiency of asynchronous patterns in JavaScript remains an unexplored and promising area for research. Table 1 summarizes the key related studies discussed in this literature review, highlighting their focus areas and methodology.

Table 1. Summary of Related Research Studies in JavaScript Asynchronous Programming and Energy Efficiency

Study	**Language Features**	**Study Focus**	**Methodology**	**Metrics Used**
Primary Studies				
Selakovic & Pradel (2016)	API usage, DOM operations	Performance	Static analysis	Execution time, memory usage

Continued on next page

Table 1 (continued)

Study	Language Features	Study Focus	Methodology	Metrics Used
Gokhale et al. (2021)	Sync vs async operations	Performance	Automated refactoring	Execution time, scalability, throughput
Patrou et al. (2021)	Event loop, async I/O	Performance	Empirical analysis	CPU usage, memory consumption, I/O throughput
Pereira et al. (2021)	Core language features	Energy efficiency	Cross-language comparison	Energy consumption, execution time
Turcotte et al. (2022)	Promise chains	Performance, energy	Tool-based analysis	Anti-pattern detection, energy consumption
Secondary Studies				
Fritz & Zhao (2017)	Async patterns	Developer experience	Syntax design	Productivity, readability
Alimadad (2018)	Async patterns, error handling	Code quality	Pattern analysis	Anti-pattern frequency, structure
Luong & Le (2019)	Callback, promise, async/await	Code quality	Theoretical review	Complexity, readability
Turcotte (2023)	Promise chains, async/await	Code quality	Static analysis	Maintainability metrics

3 Methodology

This chapter presents our methodology. It begins with a review of the evolution of asynchronous patterns in JavaScript, then outlines our research questions, and finally describes the experimental setup, protocol, and the design and implementation of test cases for asynchronous pattern comparisons.

3.1 Asynchronous Patterns Overview

This section reviews the evolution and characteristics of asynchronous patterns in JavaScript. Over time, JavaScript has developed increasingly sophisticated mechanisms to handle asynchronous operations. The evolution can be traced through three primary asynchronous patterns:

The callback pattern (Pre-2015). Initially, JavaScript developers relied on the callback pattern, where a function initiating an asynchronous task accepts another function (the callback function) as an argument. The initiating function invokes this callback function once the asynchronous task is completed. Despite predating 2015, the callback pattern remains relevant in modern JavaScript development for its simplicity in handling basic asynchronous operations, as it continues to be a practical solution in many scenarios. Although appropriate for a simpler use case, chaining multiple asynchronous operations using the callback pattern can lead to deeply nested code structures, termed "callback hell," which presented maintenance challenges, as described in the MDN Web Docs developer guide [20]. The following example, adapted from this guide, illustrates the callback pattern where a function accepts another function as an argument and executes it after completing its task:

```
function callbackTask(callback) {
    console.log("Starting operation");
    callback();
}

callbackTask(() => {
    console.log("Operation completed");
});
```

The code shows how a callback function is passed to `callbackTask` and executed after the initial operation. When multiple operations need to be coordinated, the callback pattern results in nested structures, as demonstrated in this second example, also derived from the developer guide:

```
callbackTask(() => {
    console.log("First operation completed");
    callbackTask(() => {
        console.log("Second operation completed");
        callbackTask(() => {
            console.log("Third operation completed");
        });
    });
});
```

This example illustrates how nested function calls in the callback pattern create deeply indented code structures, representing the "callback hell" issue that makes code more challenging to maintain and understand.

The promise pattern (ECMAScript 2015). The ECMAScript® 2015 standard [32] introduced the promise pattern to address the challenges of the callback pattern. In the promise pattern, a `Promise` represents an asynchronous operation with three states: `pending`, `fulfilled`, or `rejected`. This state-driven model enabled more elegant operation chaining and error handling, reducing the complexity of using the callback pattern. The following two simplified examples, which we created based on the ECMAScript® 2015 documentation, demonstrate implementation of the same asynchronous operation using the promise pattern:

```
function promiseTask() {
    return new Promise(resolve => {
        console.log("Starting operation");
        resolve();
    });
}

promiseTask().then(() => {
    console.log("Operation completed");
});
```

Here, `promiseTask` returns a `Promise` that resolves upon operation completion. The `.then()` method provides a cleaner way to handle the operation's completion compared to the callback pattern. When chaining multiple operations, using the promise pattern maintains a more manageable structure:

```
promiseTask()
    .then(() => {
        console.log("First operation completed");
        return promiseTask();
    })
    .then(() => {
        console.log("Second operation completed");
        return promiseTask();
    })
    .then(() => {
        console.log("Third operation completed");
    });
```

The flat structure of `Promise` chaining contributes to improved readability and error handling compared to the nested code structure of the callback pattern.

The async/await pattern (ECMAScript 2017). Building on the promise pattern, the ECMAScript® 2017 standard [33] introduced the async/await pattern with keywords `async` and `await`. The `async` keyword declares a function that returns a `Promise`, while the `await` keyword pauses execution until a `Promise` resolves, allowing asynchronous operations to be written with synchronous-looking syntax. While the async/await pattern is a syntactic sugar over the promise pattern, it enhances code readability and maintainability. The following two examples are created based on the ECMAScript® 2017 documentation:

```
async function asyncTask() {
    console.log("Starting operation");
    return "Operation completed";
}

(async () => {
    const result = await asyncTask();
    console.log(result);
})();
```

When handling multiple operations, the syntax of the async/await pattern maintains a synchronous-like structure:

```
(async () => {
    const result1 = await asyncTask();
    console.log("First operation completed");
    const result2 = await asyncTask();
    console.log("Second operation completed");
    const result3 = await asyncTask();
    console.log("Third operation completed");
})();
```

The sequential style makes the code flow more intuitive and easier to reason about than previous patterns.

These advancements in JavaScript have improved the developer experience by increasing the readability and maintainability of asynchronous code, as demonstrated in Luong and Canh's study [28]. However, each abstraction layer potentially adds computational overhead, so we need to investigate whether this translates into significant costs in terms of energy consumption.

3.2 Research Questions

Each new asynchronous pattern was designed to improve code maintainability. However, more primitive patterns may be more energy-efficient due to reduced complexity. With this in mind, we present the research question guiding our study:

RQ1: How do selected JavaScript asynchronous patterns compare in energy consumption?

> H_0: There is no significant difference in energy consumption between selected asynchronous patterns.
>
> H_a: There is a significant difference in energy consumption between selected asynchronous patterns.

This research question is addressed through an experiment detailed in the following sections.

3.3 Experimental Setup

We begin by configuring our experimental setup. This phase involved configuring both the hardware and the software environments. Throughout the experiments, hardware and software configurations were kept constant as controlled variables to examine the impact of asynchronous patterns and execution time (the independent variables) on energy consumption (the dependent variable).

We had a machine with an Intel Core i5-1135G7 processor (11th Gen, 2.40GHz base, 4.20GHz turbo) and 16GB RAM available to us, and decided to use that for the experiments. We selected the Perf tool for energy measurements, which provides access to Intel RAPL (Running Average Power Limit) [34]. RAPL is an energy profiling tool that has been featured in several studies cited in this thesis; three studies [10, 17, 21] used it for measurements, and two other studies [18, 35] included it in energy profiler comparisons.

In one of these comparison studies, Jay et al. [35] reported that RAPL estimates energy consumption using direct measurements from the processor's voltage regulators and monitoring hardware events. It provides energy estimates for specific hardware domains, including the CPU package (PKG), memory (DRAM), and the entire SoC (PSys). These energy consumption reports are accessible via MSR (model-specific registers), which store values in microjoules consumed since processor startup. Their study compared energy profilers that provide easier access to the RAPL interface without directly dealing with MSR registers. These tools include PowerAPI, Scaphandre, Energy Scope, and Perf. Based on these findings, we selected the Perf tool for our measurements as it was compatible with Intel hardware and had good features. The Perf tool supports high sampling frequency by default, which is necessary for measuring fairly quick-running asynchronous operations. With low measurement overhead (under 1%) and the ability to measure larger hardware domains than just the CPU, the Perf tool offered exactly what we needed. The researchers also rated the Perf tool favorably for ease of use and documentation. Installation and usage proved to be straightforward. As the Perf tool is limited to Linux-based systems, Ubuntu 24.04.1 LTS [36] was installed as the operating system.

For our JavaScript runtime environment, we used Node.js [6] v18.19.1, built on the

[6]Node.js is a runtime environment that allows JavaScript code to be executed outside of browsers [37].

V8 engine [7], which handles JavaScript parsing, compilation, execution, and memory management. We selected Node.js for its widespread industry adoption and ease of use [26]. With the experimental environment established, we next needed to ensure consistent test conditions.

3.4 Experimental Protocol

We developed an experimental protocol to ensure consistent conditions and minimize measurement variations. The protocol included the following steps:

We set the display brightness to minimum to reduce power consumption. We allowed the machine to rest for 5 minutes before each measurement to prevent thermal throttling [8]. We disabled unnecessary background processes using Ubuntu's System Monitor, including applications, browsers, backup services, and media players. We connected the test machine to an isolated network to minimize interference. We added code-level controls by running our tests with the `-expose-gc` flag, triggering `global.gc()` garbage collection cycle before each test, and clearing module caches between measurements.

Following these steps before each measurement ensured consistent starting conditions on all tests. With our experimental protocol in place, we will look at how we designed our test cases and the measurement process.

3.5 Test Case Design

To describe our test cases, we define the following terminology.

Pattern Function: A JavaScript function that implements one of the three asynchronous patterns to perform a task.

Version: A complete implementation file containing a pattern function along with necessary setup code. Each version represents a combination of a test case and an asynchronous pattern.

Measurement Script: A script using the Perf tool to execute a version and produce a result file.

[7] V8 is a JavaScript engine that implements ECMAScript and powers Node.js [38].

[8] Thermal throttling is a mechanism where processors reduce performance to manage heat buildup.

Result File: A timestamped file with output generated by the Perf tool.

Measurement: The energy consumption metric extracted from a result file.

Test Run: A single execution of the measurement script for one version.

`runTest` Function: Utility function within each version that executes a pattern function.

These terms are further explained in context throughout the following sections, but readers can refer back to these definitions at any point for clarity. The following subsections describe the test cases.

3.5.1 Test Case Overview

Two test cases were designed to compare the energy consumption of selected asynchronous patterns: Test Case 1 involving HTTP requests (TC1) and Test Case 2 involving file writing (TC2). HTTP was chosen for TC1 because it is the most common protocol for requests in web applications. Recognizing that HTTP requests commonly use different HTTP methods, we structured the test case to use both GET and POST requests. In this study, "HTTP requests" refers exclusively to HTTP GET and HTTP POST requests. The two test cases represent common asynchronous operations in web applications, involving data transfer via a network and writing data to the local file system. Both test cases addressed the research question, so we could analyze how energy consumption might vary depending on the type of asynchronous operation being performed.

For each test case, multiple implementations were created, referred to as "versions." Each version consists of a JavaScript file containing a "pattern function" that implements one of the three asynchronous patterns to perform the specific task. Each pattern function was designed to provide identical functionality across versions of the same test case, with the overall code structure maintained consistently, accounting only for necessary syntax differences specific to each asynchronous pattern.

In total, nine distinct versions were developed, representing the combination of context (HTTP request or file write), asynchronous pattern (callback, promise, async/await), and HTTP method (GET or POST) where applicable, as shown in Figure 1.

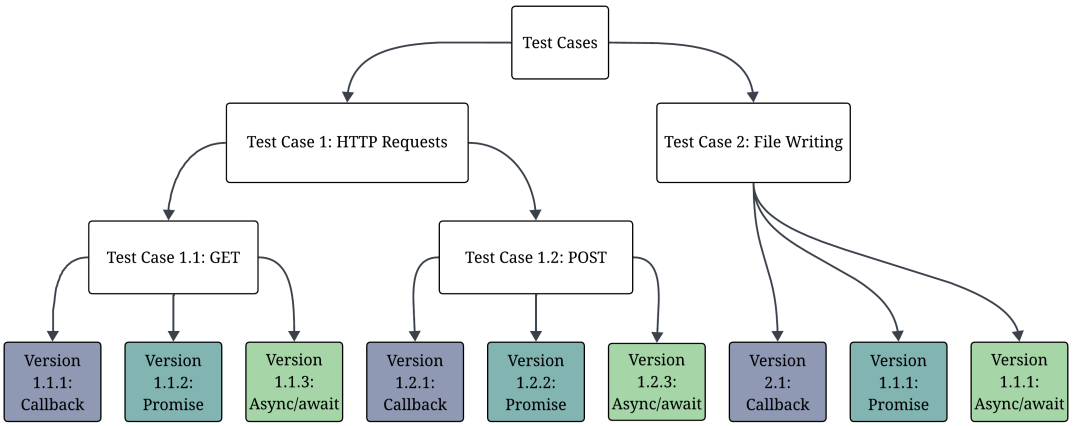

Figure 1. Overview of test case versions.

3.5.2 Test Case 1: HTTP Requests

HTTP requests are a common way to transfer data between clients and servers in web applications. This process is often handled asynchronously to ensure the user interface remains responsive. A typical example is when a social media user scrolls through their feed, and the application retrieves data asynchronously using HTTP requests. In this scenario, the user's device acts as the client initiating these requests for content from the platform's remote servers, delivering the requested data. Both server and client components are required to simulate this interaction for a test case.

For the server component, we set up a local Express.js (v4.18.2, a minimal web application framework for Node.js) [39] server running on port 3000. Our server exposes two endpoints: a GET request endpoint at `/test` and a POST request endpoint at `/test`. The payloads used are JSON[9], which we chose over other popular web data formats like XML or plain strings because it integrates well with JavaScript. Both endpoints handle 1.5KB payloads: the GET request endpoint serves this data when requested, and the POST request endpoint accepts it from clients. We set this payload size to match real-world usage, as Rodriguez et al. [40] found that typical JSON payloads in API responses had a median size of 1545 bytes when analyzing actual internet traffic. We created mock JSON data of this size for our server.

[9]JSON (JavaScript Object Notation) is a lightweight, text-based, human-readable data-interchange format.

For the client component, each version acts as a client, with its pattern function performing an HTTP request to either the GET or POST request endpoint using Node.js's built-in http module. All pattern functions rely on shared utility functions we developed to prepare requests: createGetRequestConfiguration for GET requests and createPostRequestConfiguration for POST requests. Both include a fixed delay to simulate latency, which ensures that all requests take the same time to complete regardless of when they are sent, effectively mitigating network timing variations. The value of 150ms reflects the upper bound of typical successful API response times observed in a 3-month-long benchmark study of popular APIs [41]. Additional processing, such as encryption or compression, was excluded to avoid introducing variables that could obscure differences.

Here is a list of all the versions created for Test Case 1:

- **Version 1.1.1 – Callback:** Implements the pattern function:
  `function callbackGET(callback)`
  that uses Node's `http.request` method with GET to retrieve data from the local server.
- **Version 1.1.2 – Promise:** Implements the pattern function:
  `function promiseGET()`
  that uses Node's `http.request` method with GET to retrieve data from the local server, returning a `Promise` that resolves with the retrieved data.
- **Version 1.1.3 – Async/await:** Implements the pattern function:
  `async function asyncAwaitGET()`
  which uses Node's `http.request` method with GET to retrieve data from the local server, enabling `await` syntax for sequential asynchronous flow.
- **Version 1.2.1 – Callback:** Implements the pattern function:
  `function callbackPOST(callback)`
  that uses Node's `http.request` method with POST to send data to the local server.
- **Version 1.2.2 – Promise:** Implements the pattern function:
  `function promisePOST()`
  that uses Node's `http.request` method with POST to send data to the local server, returning a `Promise` that resolves with the server's response.

- **Version 1.2.3 – Async/await:** Implements the pattern function:
  `async function asyncAwaitPOST()`
  which uses Node's `http.request` method with POST to send data to the local server, enabling `await` syntax for sequential asynchronous flow.

3.5.3 Test Case 2: File Writing

Writing to a file is another type of asynchronous task in server environments. Examples include writing error stack traces for debugging or logging client/server requests. This must be done asynchronously to allow the server to continue handling other requests without interruption.

This test case focuses on simulating server error logging. This scenario requires only a server component. Each version represents a server with a pattern function that is responsible for writing the same 1.5KB mock data used in TC1 to a log file in a dedicated directory. Each pattern function implements file writing using Node.js's native `fs` module, which supports the three asynchronous patterns. The file system cache was cleared, and previous log files were removed before each test run to ensure consistent measurements.

Here is a list of all the versions created for Test Case 2:

- **Version 2.1 – Callback:** Implements the pattern function: `function callbackFileWrite(callback)` that uses Node's `fs.writeFile`.
- **Version 2.2 – Promise:** Implements the pattern function: `function promiseFileWrite()` that uses `fs.writeFile`, returning a `Promise` that resolves with the file content.
- **Version 2.3 – Async/await:** Implements the pattern function: `async function asyncAwaitFileWrite()` that uses `fs.writeFile`, allowing `await` syntax for linear asynchronous flow.

3.6 Conducting Measurements

Having created nine distinct versions, we needed a consistent way to test them. We created the `runTest` function to standardize our testing across all versions. This function

accepts a pattern function as its sole parameter. It executes it 10 consecutive times, which reflects typical web application usage, where users perform multiple actions in sequence rather than just one action before closing the application. This also helps our measurement tool properly detect the energy consumption, as single executions of pattern functions may be too brief to measure, and the overhead of executing a version would mask differences.

The `runTest` function was created in a shared utility file that all versions import. In the main function of each version, `runTest` is called with that version's pattern function as the parameter. This structure ensured that all pattern functions were tested the same way. For executing each version and measuring its energy use, we created a command using Node.js for execution and the Perf tool as a wrapper for energy measurement, based on this template:

```
sudo perf stat -a -e power/energy-psys/ node --expose-gc <version.js>
```

We used the system-wide `-a` flag along with `-e power/energy-psys/` to capture energy consumption across the entire system beyond just the CPU. We developed measurement scripts with the help of Microsoft Copilot (2025) to capture the raw Perf tool output from executing the command into result files. Each execution of this measurement script constitutes a test run, and each test run produces a result file. A measurement in our study consists of energy consumption (in Joules) extracted from these result files. The complete resources, including measurement scripts, versions, and result files, are available in the repository detailed in the Appendix I.

We performed 30 test runs for each version to gather enough measurements. Any number of measurements we take can only be a sample of all possible measurements. Still, the Central Limit Theorem tells us that with a sufficient sample size, the distribution of sample means will approximate a normal distribution regardless of the underlying population distribution. This statistical principle is well-established in the field; Kwak and Kim [42], one of many works discussing this concept, notes that a sample size of 30 is generally considered the threshold at which the sampling distribution becomes equivalent to the normal distribution. This means we can run 30 tests and be confident that our results accurately represent the system's typical performance.

3.7 Analysis Tools and Methods

After collecting measurements, we processed them for analysis. Each test run produced a result file stored in a dedicated directory. We used a script from Microsoft Copilot (2025) to extract and combine measurements from all 30 result files per version into CSV files.

We used Python for the analysis, working within a Jupyter Notebook [43] hosted on Google Colab [44]. This setup was chosen because it offered a pre-installed Python setup with common data analysis libraries and provided a convenient way to combine code and text for documenting the analysis. The Jupyter Notebook used for this analysis can be found in the repository (see Appendix I).

Our analysis consisted of multiple steps. First, we loaded the CSV data and organized it by test case and pattern. We then analyze descriptive statistics for each pattern, e.g., means and medians. These descriptive statistics provided a first overview of the data and initial ranking of the patterns by energy efficiency.

After reviewing descriptive statistics, we needed to conduct statistical tests to determine whether the observed differences between patterns were statistically significant. To choose the appropriate tests, we first needed to assess the normality of our data distribution, as many parametric statistical tests assume that data follows a normal distribution. We selected the Shapiro-Wilk test [45] with a significance level of $\alpha = 0.05$ to evaluate normality. This test was chosen because it has high statistical power in detecting deviations from normality, particularly for smaller sample sizes like ours ($n = 30$ per group). The Shapiro-Wilk test compares the observed data distribution to an expected normal distribution, with the null hypothesis being that the data is normally distributed.

The normality test results showed that most of our data was not normally distributed. Therefore, we used the Kruskal-Wallis H test, described by Kruskal and Wallis [46]. Based on that article [46], this non-parametric statistical test is designed for comparing two or more independent samples when the assumption of normality is not met. The test evaluates the null hypothesis that all samples come from identical populations, which means if the result is significant ($p < 0.05$), at least one sample differs meaningfully from the others. A significant result in our case indicates that energy consumption measurements from at least one asynchronous pattern differ statistically from those of the others.

We found a statistically significant overall difference among patterns using the

Kruskal-Wallis test. Since the test only confirmed an overall difference but did not specify between which pairs, we used post-hoc Dunn's tests [47] to pinpoint these specific pairwise differences. This test performs multiple pairwise comparisons (e.g., three comparisons for three patterns). Since each comparison carries a risk of a false positive, performing several pushes the overall probability of making any such error above $\alpha = 0.05$. To maintain the $\alpha = 0.05$ overall error rate in all comparisons, we used the Bonferroni correction [48], which adjusts the significance threshold for each test to be more conservative ($\frac{\alpha}{3}$), thereby controlling the overall risk of a false positive. The described methods were applied to determine whether statistically significant differences existed between the energy consumption measurements for each pattern (RQ1).

4 Results and Analysis

This chapter presents the results from our experiment, focusing on the primary research question regarding energy consumption (RQ1). We first present the statistical test results, then offer our conclusion on the hypothesis.

4.1 Energy Consumption of Selected Asynchronous Patterns (RQ1)

We revisit our previously stated research question. **RQ1:** How do selected JavaScript asynchronous patterns compare in energy consumption?

> H_0: There is no significant difference in energy consumption between selected asynchronous patterns.
>
> H_a: There is a significant difference in energy consumption between selected asynchronous patterns.

4.1.1 Test Case 1 results

We begin by examining the descriptive statistics of measurements for TC1, as presented in Tables 2 and 3. For GET requests, the callback pattern shows the lowest mean energy consumption (3.180 J), followed by the promise pattern (3.370 J), and then the async/await pattern (3.394 J). Both mean and median values show the same ascending order of energy consumption of the asynchronous patterns: callback, promise, then async/await.

Table 2. Energy Consumption for GET Requests (TC1) (Joules)

Pattern	Mean	Median	Std Dev	Min	Max
Callback	3.180	3.080	0.428	2.610	4.550
Promise	3.370	3.180	0.728	2.820	6.660
Async/await	3.394	3.305	0.364	2.850	4.620

Table 3. Energy Consumption for POST Requests (TC1) (Joules)

Pattern	Mean	Median	Std Dev	Min	Max
Callback	3.187	3.220	0.202	2.710	3.470
Promise	3.270	3.310	0.336	2.820	4.510
Async/await	3.383	3.340	0.297	2.850	4.190

A similar trend is seen for POST requests: the callback pattern shows the lowest mean energy consumption (3.187 J), followed by the promise pattern (3.270 J), and then the async/await pattern (3.383 J). Energy consumption appears more consistent across POST requests than GET requests, as evidenced by the lower standard deviations. The distribution of energy consumption measurements across the three asynchronous patterns for both GET and POST requests in TC1 is illustrated in Figure 2.

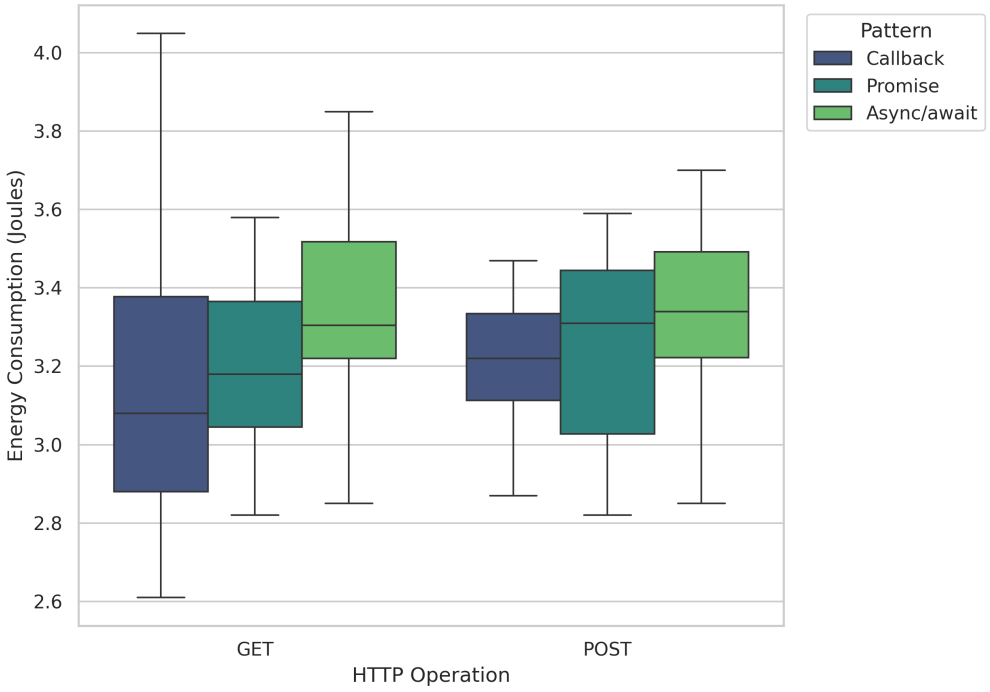

Figure 2. Energy consumption of asynchronous patterns for TC1.

We first assessed data normality to determine whether these observed differences are statistically significant. The Shapiro-Wilk test ($\alpha = 0.05$) was used, and the results (Table 4) indicate that most energy consumption measurements for TC1 deviated significantly

from a normal distribution ($p < 0.05$). This non-normality was observed across all patterns for both GET and POST requests.

Table 4. Shapiro-Wilk Normality Test Results for Energy Consumption (p-values) (TC1)

Pattern	GET	POST
Callback	0.007*	0.071
Promise	<0.001*	0.001*
Async/await	0.008*	0.018*

Note: * indicates $p < 0.05$ (significant deviation from normality).

Given the non-normality, the Kruskal-Wallis test was conducted to compare the energy consumption distributions among the three patterns. Table 5 shows the results of this test ($\alpha = 0.05$). For HTTP GET requests, we obtained $p = 0.030$, while for HTTP POST requests, we obtained $p = 0.035$. Both p-values are below our significance threshold, indicating statistically significant differences in energy consumption among the three asynchronous patterns for TC1. These significant results reject the null hypothesis that all selected asynchronous patterns have equal energy consumption distributions, but do not identify which specific pairs of patterns differ significantly.

Table 5. Kruskal-Wallis Test Results for Energy Consumption (TC1)

Method	H Statistic	p-value
GET	7.028	0.030
POST	6.732	0.035

Following the significant Kruskal-Wallis results, Dunn's post-hoc tests ($\alpha = 0.05$) with Bonferroni correction ($\frac{\alpha}{3}$) were performed to identify specific pairwise differences. The results in Table 6 reveal that for GET requests, a significant difference exists between the async/await pattern and the callback pattern ($p = 0.027$), but not between other pattern pairs (callback vs. promise, $p = 1.000$; async/await vs. promise, $p = 0.282$). Similarly, for POST requests, a significant difference exists between the async/await pattern and the callback pattern ($p = 0.030$), but not between other pattern pairs (callback vs. promise, $p = 0.877$; async/await vs. promise, $p = 0.380$).

Table 6. Dunn's Post-Hoc Test Results for Energy Consumption (p-values) (TC1)

Request	Comparison Pair	Async/await	Callback	Promise
GET	Async/await	-	0.027*	0.282
	Callback	0.027*	-	1.000
	Promise	0.282	1.000	-
POST	Async/await	-	0.030*	0.380
	Callback	0.030*	-	0.877
	Promise	0.380	0.877	-

Note: * indicates $p < 0.05$ (significant difference).

The results show that the callback pattern was significantly more energy-efficient than the async/await pattern for TC1. The promise pattern's energy consumption was between the other two, but its difference from the callback or async/await pattern was not statistically significant.

4.1.2 Test Case 2 results

For TC2, Table 7 presents the descriptive statistics for each asynchronous pattern. For this test case, mean energy consumption values are much closer across patterns: callback (3.207 J), promise (3.301 J), and async/await (3.268 J). Notably, while the callback pattern has the lowest mean energy consumption, the promise pattern has the lowest median (3.105 J), suggesting some outliers affecting the mean of the promise pattern measurements. The range between minimum and maximum values is widest for the promise pattern (2.630 J to 5.300 J), further supporting the observation of greater variability.

Table 7. Energy Consumption for File Writing (TC2) (Joules)

Pattern	Mean	Median	Std Dev	Min	Max
Callback	3.207	3.190	0.360	2.570	3.930
Promise	3.301	3.105	0.585	2.630	5.300
Async/await	3.268	3.350	0.316	2.750	3.830

The distribution of energy consumption for the three asynchronous patterns in TC2 is shown in Figure 3.

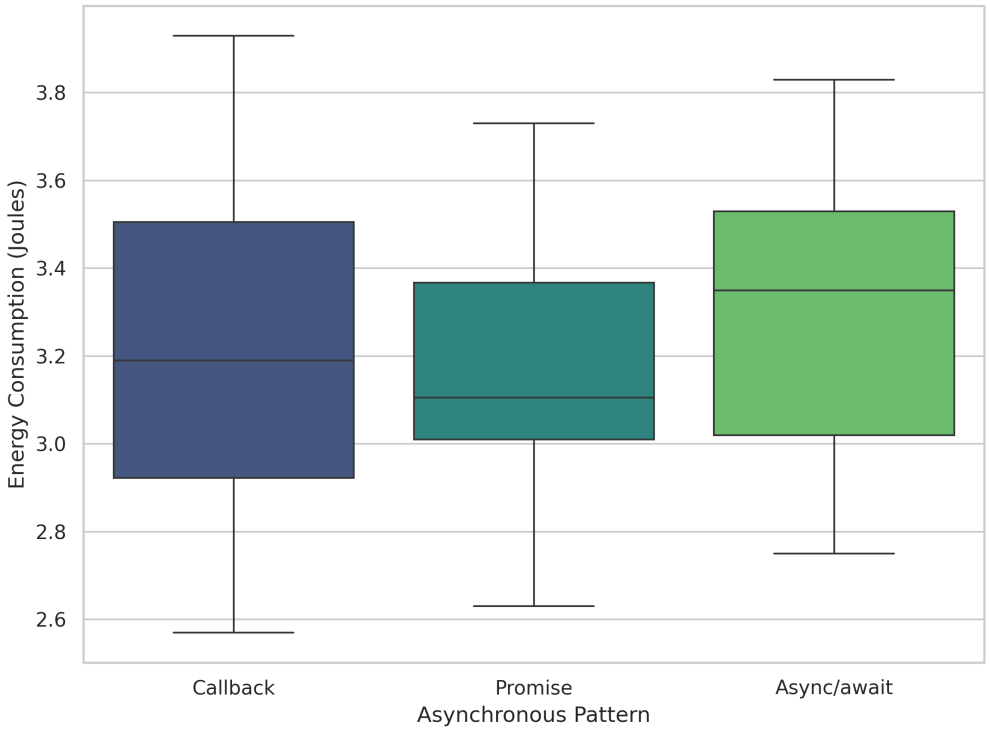

Figure 3. Energy consumption of asynchronous patterns for TC2.

As with TC1, we first assessed the normality of our data to determine the appropriate statistical methods. Table 8 shows that for TC2, the energy consumption measurements for the callback pattern ($p = 0.551$) and the async/await ($p = 0.271$) pattern did not significantly deviate from normality, as their p-values exceed our threshold of $\alpha = 0.05$. However, the promise pattern data did show significant deviation from normality ($p < 0.001$).

Table 8. Shapiro-Wilk Normality Test Results for Energy Consumption (p-values) (TC2)

Pattern	p-value
Callback	0.551
Promise	<0.001*
Async/await	0.271

Note: * indicates $p < 0.05$ (significant deviation from normality).

Due to non-normality in the promise pattern measurements group, we remained consistent and used the non-parametric Kruskal-Wallis test for this test case as well.

The Kruskal-Wallis test result for TC2 showed $p = 0.703$. This p-value is well above our significance threshold ($\alpha = 0.05$), indicating no statistically significant differences in energy consumption among the three asynchronous patterns for TC2. This means we fail to reject the null hypothesis that the distribution of energy consumption is the same across all three patterns. The large p-value ($p = 0.703$) suggests that any observed differences are likely due to random variation.

Given this non-significant result in the Kruskal-Wallis test, we did not proceed with post-hoc tests for this test case, as post-hoc tests are only warranted when the initial test indicates significant differences among the groups.

4.1.3 Conclusion for RQ1

We can draw the following conclusions regarding our research question based on our statistical analysis.

For TC1, the null hypothesis (H_0), stating no difference in energy consumption between the asynchronous patterns, was rejected in favor of the alternative hypothesis (H_a), which suggests statistically significant differences exist among the patterns for this test case. Subsequent post-hoc analysis revealed a statistically significant difference in energy consumption between the callback and the async/await patterns. No statistically significant differences were detected for other pairs.

For TC2, we failed to reject the null hypothesis (H_0). This indicates that no statistically significant differences in energy consumption were detected between the three asynchronous patterns for this test case.

4.2 Discussion of Findings

Statistical test results suggest significant energy consumption differences between asynchronous patterns within our controlled scenarios. Even so, selecting a pattern for application requires consideration beyond statistical significance (p-values). This section discusses the real-world relevance of the findings and other factors influencing the adoption of these patterns in actual use.

4.2.1 Interpreting the Results

Results differed between the two test cases:

- **HTTP Requests (TC1):** A statistically significant difference in energy consumption was observed under our test conditions, where the async/await pattern consumed on average 6.75% more energy for GET requests and 6.15% more energy for POST requests compared to the callback pattern.
- **File Writing (TC2):** No significant difference in energy consumption was found between the patterns for this task in our controlled environment.

Results for the promise pattern generally fell between the callback and the async/await patterns in our tests. However, the differences were not always statistically significant compared to the other two patterns in the same test.

Just because a difference is statistically significant under our controlled settings does not mean it is essential in practice. We need to think about how much these differences matter. The finding that the callback pattern uses on average $\sim$6-7% less energy in our controlled settings for HTTP requests (TC1) could be important in some cases. For example, in large applications, such as those run by Meta Platforms, which handle around 140 billion messages daily [49] (many likely involving some form of requests), this could add up to substantial energy savings. Also, lower energy consumption could help the battery last longer, improving user experience for applications running on phones or other battery-powered devices.

4.2.2 Qualitative Factors

When choosing an asynchronous pattern, developers consider factors beyond energy consumption benchmarked in studies like this. Code readability and maintainability are

considerations for developers. The callback pattern is the most basic but can lead to messy code with many nested functions, making the code difficult to understand, debug, and maintain.

The promise pattern was introduced to address these issues by providing a more structured approach to chaining asynchronous operations. The `async/await` pattern further improves readability by allowing asynchronous code to resemble synchronous code. Many developers perceive these newer patterns as more straightforward to read and manage.

Error handling is also simpler with the newer patterns. The `Promise` object in the promise pattern provides a `.catch()` method for handling errors in a chain, and the keywords `async` and `await` of the async/await pattern allow using standard `try` and `catch()` blocks, similar to synchronous code. These features are often considered clearer and more reliable than checking errors in the callback pattern, and can lead to fewer bugs caused by unhandled errors.

Developer preference and productivity are additional factors influencing pattern selection. The promise and the async/await pattern are commonly featured in modern tutorials, examples, and libraries, making developers' learning and collaboration easier.

The concept that simpler patterns may have less overhead is relevant. The results, particularly the energy difference in TC1, suggest that the features enhancing the usability of the promise and the async/await patterns may introduce slightly more overhead compared to the more direct callback pattern. This implies a potential trade-off between the usability features of the promise and the async/await patterns and the efficiency of the callback pattern, a factor for developers to consider.

4.2.3 Concluding Discussion

Asynchronous pattern selection depends on the specific context and involves balancing improvements in energy consumption against practical considerations of code development and maintenance. This study found that the callback pattern has certain advantages depending on the test case. It showed significantly lower energy consumption for HTTP requests (TC1). However, these modest benefits should be weighed against the advantages of the promise and the async/await patterns in terms of code organization, error handling, and development ease.

In cases where marginal energy savings for network tasks (approximately 6-7%) are critically important, using the callback pattern could be considered, provided strategies are in place to manage potential code complexity.

For most typical web development contexts, the benefits of using the async/await pattern, which contribute to improved code readability, reduced error potential, and faster development, are likely more impactful than the small energy consumption differences observed. Developers should evaluate these trade-offs based on project requirements and priorities, considering both technical performance and factors related to code quality and development efficiency.

4.2.4 Limitations and Threats to Validity

This section discusses potential limitations and threats to the validity of our study, along with the measures taken to mitigate them.

Internal Validity:

- *Measurement Precision:* We used the Perf tool, which accesses Intel RAPL, to estimate energy consumption. While the Perf tool does not provide direct physical power measurements, it is consistent enough for comparative analysis. We further improved reliability by conducting multiple measurements and using statistical methods to account for variance.
- *System State:* Background processes and system services might affect energy measurements. We minimize this by following the experimental protocol.

External Validity:

- *Hardware:* Results were obtained on laptop hardware, which differs from server environments typical for high-traffic applications. Energy consumption may vary across different processors and architectures.
- *Software:* Results are specific to the software configuration used and may vary with different JavaScript engines, runtime versions, modules, or energy measurement tools. Variations in software components could affect measurement outcomes.
- *Simplified Test Cases:* Test cases are simplified to make findings applicable to many applications. More complex, specific test cases would limit relevance since

real-world applications vary. While this approach might not capture all real-world complexities, it allows our results to remain applicable across various scenarios.

Construct Validity:

- *Fixed latency:* Using a fixed latency to simulate HTTP requests does not replicate the dynamic nature of real-world networks. However, it provided a controlled environment necessary for comparisons.

Conclusion Validity:

- *Sample Size:* A sample size of 30 measurements per configuration was used to ensure sufficient statistical power.
- *Statistical Analysis:* Appropriate statistical tests were used based on data, and a significance level was included to support conclusions.

Applicability of Findings

The findings of this thesis apply to many JavaScript applications. Large-scale web applications with billions of daily users benefit most, as even small per-user energy reductions can lead to significant overall savings. In addition, applications that need moderate energy improvements to boost performance or reduce energy costs can benefit by implementing the energy-efficient asynchronous pattern identified in this research.

However, these findings are not universal for all JavaScript applications. The energy savings discussed may be negligible for smaller-scale applications with limited users or infrequent activity. Additionally, when considering truly energy-critical systems where every last bit of power efficiency matters, JavaScript may not be the optimal choice, and using proven faster languages like C++ likely remains necessary. These limitations and threats to validity are acknowledged in the interpretation of results and should be considered when generalizing the findings to other contexts.

5 Conclusion

This thesis investigated the energy consumption of JavaScript's asynchronous patterns (callback, promise, async/await). The study was motivated by the increasing energy consumption of the ICT sector and the prevalence of JavaScript in web development. The aim was to provide data on potential energy efficiency trade-offs among these patterns.

The controlled experiments were conducted using the Node.js runtime environment. The test cases simulated asynchronous tasks: HTTP requests (GET and POST with simulated latency) and file writing. Energy consumption was measured using the Perf tool. Measurements were taken from 30 test runs for each pattern within each test case, followed by statistical analysis.

The key findings revealed statistically significant differences in the energy consumption for HTTP requests (TC1), where the callback pattern consumed on average 6-7% less energy than async/await. For file writing (TC2), no significant differences in energy consumption were observed. The promise pattern generally produced results between the callback and the async/await patterns, although these differences were often not statistically significant against both alternatives in the tests.

The primary contribution is comparative data on the energy consumption of JavaScript's asynchronous patterns under our controlled conditions. The results suggest that the callback pattern can offer minor performance benefits (energy for HTTP requests, time for file writing) in the tested scenarios. However, these advantages are relatively small. The practical implication is that for most web development projects, the benefits of the promise and async/await patterns, specifically improved code readability, maintainability, error handling, and developer productivity, likely outweigh these minor energy efficiency gains in our controlled experiment. The selection of patterns should align with the project priorities. If marginal energy savings are critical (e.g., in large-scale applications or energy-constrained environments), the callback pattern may be considered, provided that associated complexity is managed. For most applications, the advantages of modern asynchronous patterns are substantial.

The study has limitations. Experiments were conducted on specific hardware, and results may vary on other architectures. Test cases were simplified and may not capture all real-world complexities. The Perf tool provides energy estimations, not direct hardware measurements.

Future research could explore several directions. Repeating experiments on other hardware platforms would improve generalizability. Investigating the impact of varying network conditions and payload sizes could provide a more detailed understanding. Using higher-resolution profiling tools might help explain observed energy differences. Expanding the study to include more complex asynchronous workflows or different JavaScript runtimes could offer further insights.

List of References

[1] Andrae A. 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*, 2020, Vol. 15, pp. 50–59.

[2] Malmodin J., Lövehagen N., Bergmark P., Lundén D. ICT sector electricity consumption and greenhouse gas emissions – 2020 outcome. *Telecommunications Policy*, 2024, Vol. 48, No. 3.

[3] Rozite V., Bertoli E., Reidenbach B. *Data centres and data transmission networks*. Paris: International Energy Agency, 2023. https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks (11.11.2024).

[4] Freitag C., Berners-Lee M., Widdicks K., Knowles B., Blair G. S., Friday A. The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. *Patterns*, 2021, Vol. 2, No. 9.

[5] United Nations General Assembly. *Transforming our world: the 2030 Agenda for Sustainable Development*. Resolution A/RES/70/1. New York: United Nations, 2015. https://sdgs.un.org/2030agenda (23.02.2025).

[6] Waldrop M. M. The chips are down for Moore's law. *Nature*, 2016, Vol. 530, No. 7589, pp. 144–147.

[7] Hennessy J. L., Patterson D. A. A new golden age for computer architecture. *Communications of the ACM*, 2019, Vol. 62, No. 2, pp. 48–60.

[8] Pinto G., Castor F. Energy efficiency: a new concern for application software developers. *Communications of the ACM*, 2017, Vol. 60, No. 12, pp. 68–75.

[9] Dornauer B., Felderer M. Energy-saving strategies for mobile web apps and their measurement: Results from a decade of research. *IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Melbourne: IEEE, 2023, pp. 75–86.

[10] Pereira R., Couto M., Ribeiro F., Rua R., Cunha J., Fernandes J., Saraiva J. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 2021, Vol. 205.

[11] Stack Overflow. *Stack Overflow developer survey 2023*. 2023. https://survey.stackoverflow.co/2023/ (19.12.2024).

[12] GitHub, Inc. *State of the Octoverse*. 2024. https://github.blog/news-insights/octoverse/octoverse-2024 (19.12.2024).

[13] GitHub, Inc. About GitHub: Let's build from here. 2025. https://github.com/about (23.04.2025).

[14] W3Techs. Usage statistics of client-side programming languages for websites. 2024. https://w3techs.com/technologies/overview/client_side_language (18.12.2024).

[15] Singh P., Srivastava M., Kansal M., Singh A. P., Chauhan A., Gaur A. A comparative analysis of modern frontend frameworks for building large-scale web applications. *International Conference on Disruptive Technologies (ICDT)*. Greater Noida: IEEE, 2023, pp. 531–535.

[16] Meta Platforms, Inc. *Meta reports first quarter 2025 results*. 2025. https://investor.atmeta.com/investor-events/event-details/2025/Q1-2025-Earnings-Call/default.aspx (10.05.2025).

[17] Pereira R., Couto M., Ribeiro F., Rua R., Cunha J., Fernandes J. P., Saraiva J. Energy efficiency across programming languages: how do energy, time, and memory relate?. *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. New York: Association for Computing Machinery, 2017, pp. 256–267.

[18] Georgiou S., Rizou S., Spinellis D. Software development lifecycle for energy efficiency: Techniques and tools. *ACM Computing Surveys*, 2019, Vol. 52, No. 4, pp. 1–33.

[19] Selakovic M., Pradel M. Performance issues and optimizations in JavaScript: an empirical study. *Proceedings of the 38th International Conference on Software Engineering*. New York: Association for Computing Machinery, 2016, pp. 61–72.

[20] Mozilla Developer Network. Introducing asynchronous JavaScript. s.a. https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Async_JS/Introducing (22.03.2025).

[21] Patrou M., Kent K. B., Siu J., Dawson M. Energy and runtime performance optimization of Node.js web requests. *IEEE International Conference on Cloud Engineering (IC2E)*. San Francisco: IEEE, 2021, pp. 71–82.

[22] Turcotte A., Shah M. D., Aldrich M. W., Tip F. DrAsync: identifying and visualizing anti-patterns in asynchronous JavaScript. *Proceedings of the 44th International Conference on Software Engineering*. New York: Association for Computing Machinery, 2022, pp. 774–785.

[23] Gokhale S., Turcotte A., Tip F. Automatic migration from synchronous to asynchronous JavaScript APIs. *Proceedings of the ACM on Programming Languages*, Vol. 5, No. OOPSLA, pp. 1–27.

[24] Alimadadi S., Zhong D., Madsen M., Tip F. Finding broken promises in asynchronous JavaScript programs. *Proceedings of the ACM on Programming Languages*, 2018, Vol. 2, No. OOPSLA, pp. 1–26.

[25] Microsoft (2025). *Microsoft Copilot* (GPT-4): https://copilot.microsoft.com/.

[26] Kyriakou K.-I. D., Tselikas N. D. Complementing JavaScript in high-performance Node.js and web applications with Rust and WebAssembly. *Electronics*, 2022, Vol. 11, No. 19, 3217.

[27] Turcotte A. *Optimizing asynchronous JavaScript applications*. Northeastern University, Khoury College of Computer Sciences, Ph.D. Thesis, 2023. https://doi.org/10.17760/D20560794 (16.01.2025).

[28] Luong T., Canh L. JavaScript asynchronous programming. *Hue University Journal of Science: Techniques and Technology*, 2019, No. 2A, pp. 5–16.

[29] Fritz E., Zhao T. Typing and semantics of asynchronous arrows in JavaScript. *Science of Computer Programming*, 2017, Vol. 141-142, No. 1, pp. 1–39.

[30] Cruz L., Abreu R. Performance-based guidelines for energy efficient mobile applications. *IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Argentina: IEEE, 2017, pp. 46–57.

[31] Ren J., Yuan L., Nurmi P., Wang X., Ma M., Gao L., Tang Z., Zheng J., Wang Z. Camel: Smart, adaptive energy optimization for mobile web interactions. *IEEE INFOCOM - IEEE Conference on Computer Communications*. Toronto: IEEE, 2020, pp. 119–128.

[32] ECMA International. *ECMAScript 2015 language specification*. Standard ECMA-262. 6th Edition. Geneva: ECMA International, 2015. https://262.ecma-international.org/6.0/ (11.02.2025).

[33] ECMA International. *ECMAScript 2017 language specification*. Standard ECMA-262. 8th Edition. Geneva: ECMA International, 2017. https://262.ecma-international.org/8.0/ (11.02.2025).

[34] Intel Corporation. Running average power limit energy reporting. 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html (22.03.2025).

[35] Jay M., Ostapenco V., Lefèvre L., Trystram D., Orgerie A.-C., Fichel B. An experimental comparison of software-based power meters: focus on CPU and GPU. *CCGrid 2023 - 23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*. Bangalore: IEEE, 2023, pp. 1–13.

[36] Canonical Ltd. *Ubuntu 24.04.1 LTS (Noble Numbat)*. Operating System. 2024. https://ubuntu.com/ (10.03.2025).

[37] Node.js. Introduction to Node.js. 2025. https://nodejs.org/en/learn/getting-started/introduction-to-nodejs (23.03.2025).

[38] V8. V8 JavaScript engine. 2025. https://v8.dev/ (22.03.2025).

[39] Express.js. *Express - Node.js web application framework*. 2025. https://expressjs.com/ (10.05.2025).

[40] Rodriguez C., Baez M., Daniel F., Casati F., Trabucco J., Canali L., Percannella G. REST APIs: A large-scale analysis of compliance with principles and best practices. *Lecture Notes in Computer Science Web Engineering*. Cham: Springer International Publishing, 2016, pp. 21–39.

[41] Bermbach D., Wittern E. Benchmarking web API quality – revisited. *Journal of Web Engineering*, 2020, Vol. 19, No. 5-6, pp. 603–646.

[42] Kwak S. G., Kim J. H. Central limit theorem: the cornerstone of modern statistics. *Korean Journal of Anesthesiology*, 2017, Vol. 70, No. 2, pp. 144–156.

[43] Kluyver T., Ragan-Kelley B., Pérez F., Granger B., Bussonnier M., Frederic J., Kelley K., Hamrick J., Grout J., Corlay S., Ivanov P., Avila D., Abdalla S., Willing C. Jupyter Notebooks - a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Amsterdam: IOS Press, 2016, pp. 87–90.

[44] Google (2025). *Google Colaboratory* (2025): https://colab.research.google.com/.

[45] Shapiro S.S., Wilk M. B. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 1965, Vol. 52, No. 3-4, pp. 591–611.

[46] Kruskal W. H., Wallis W. A. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 1952, Vol. 47, No. 260, pp. 583–621.

[47] Dunn O. J. Multiple comparisons using rank sums. *Technometrics*, 1964, Vol. 6, No. 3, pp. 241–252.

[48] Dunn O. J. Multiple comparisons among means. *Journal of the American Statistical Association*, 1961, Vol. 56, No. 293, pp. 52–64.

[49] Meta. Business Messaging: The Quiet Channel Revolution Across Tech. 2022. https://www.facebook.com/business/news/business-messaging-report-APAC-BCG-Meta (14.05.2025).

I Source Code

All source code used in the experiment, including the result files and the analysis file, is available in an online repository at https://github.com/akasenomm/energy-js-async-patterns.

II License

Non-exclusive licence to reproduce thesis and make thesis public

I, Artur Kasenõmm,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis "Energy Matters: Evaluating JavaScript Asynchronous Patterns for Green Development", supervised by Hina Anwar;

2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;

3. am aware of the fact that the author retains the rights specified in points 1 and 2;

4. confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Artur Kasenõmm

14/05/2025