## Summary

This project explores the energy consumption of software - more specifically it explores the energy consumption of the microservice architecture compared to the monolithic architecture. The project utilizes the well described Pet Store example to provide the solutions that are to be tested for energy consumption.

We create two implementations, one in Java and on in C#. This is in order to have a reference point for the energy consumption. In order to deploy in similar environments, and be able to easily deploy on Linux, we use Docker to containerize the solutions. To mimic users and send requests in a test plan, we utilize the Java program, JMeter as a client. JMeter is extended with *Ultimate Thread Group* that allows for ramping up the amount of requests during testing. Lastly, for the test setup, we utilize Docker Compose together with NGINX to scale the microservices. However, this was deemed to be unnecessary due to being restricted to a single test computer.

The energy consumption is measured using a Rust wrapper of Running Average Power Limit (RAPL) by Intel. To conduct the tests, a variety of shell scripts are created to automate testing and to reduce human interference. To further reduce external factors, like human interference and operating system processes, a minimal install of a server operating system is also installed. Lastly, we also follow a measurement protocol for a consistent test setup.

To gain a deep picture of the energy consumption between the two architectures, a series of experiments were conducted. One test setup consisted of the services all utilizing a single shared database. We also setup a test setup, where each microservice has access to its own database. We do this to conform to our definition of a true microservice architecture. As one of the pros of running a microservice architecture is the ability to easily scale the services by spawning new instances, we also setup a test case where we scale the services with Docker Compose and NGINX. However, these experiments were restricted as the JMeter test plan was setup to stress test the services and thus by only having 1 test computer, utilized 100% of the CPU. This meant that scaling the services only spawned more instances, with the same amount of computing power, leading to only limiting the throughput.

As our experiments with scaling services were redundant, our results only speak to the case of a microservice vs. monolithic architecture, in a non-scaling environment. From this we find that the energy per transaction for C# is 35,29% to 42% higher in the monolithic architecture compared to the microservice architecture depending on the use of individual databases for each service. For Java it is 0% to 41,18% higher. For the future work, we propose a test plan that would ensure that experiments were not restricted. Moreover, we propose more experiments and different setups that also lower the threats to validity.

AALBORG
UNIVERSITET

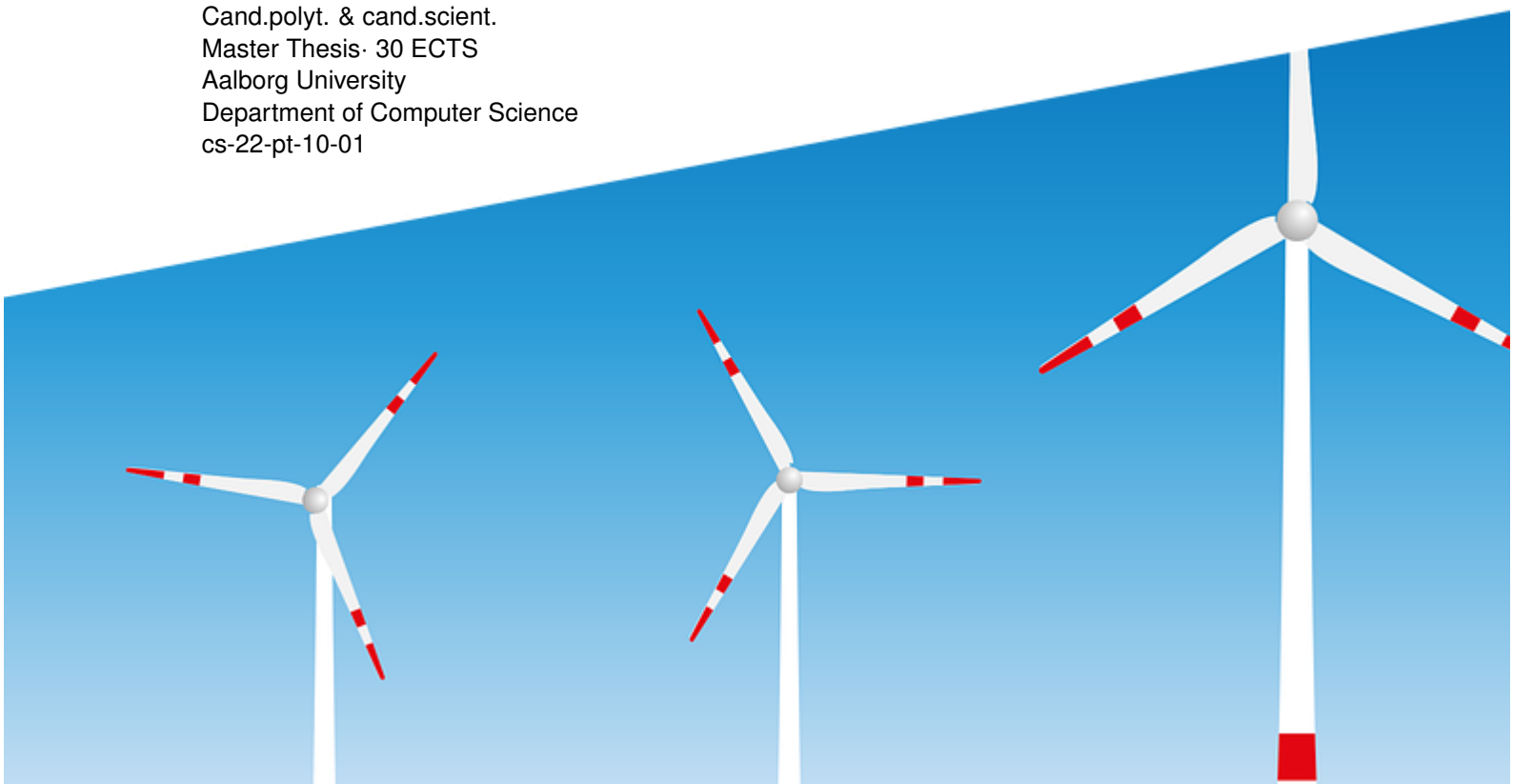
# Energy Consumption of Software Architectures

## A comparison between microservice- and monolithic architectures in C# and Java

Daniél Garrido-Y Martinez Nielsen

Kristian Theilmann Gregersen

Cand.polyt. & cand.scient.
Master Thesis· 30 ECTS
Aalborg University
Department of Computer Science
cs-22-pt-10-01



**Computer Science 10th semester**
Aalborg University
http://www.aau.dk

**AALBORG UNIVERSITY**
STUDENT REPORT

**Title:**
Energy Consumption of Software Architectures
A comparison between microservice- and monolithic architectures in C# and Java

**Theme:**
Programming Technology

**Project Period:**
Spring Semester 2010

**Project Group:**
cs-22-pt-10-01

**Participant(s):**
Daniél Garrido-Y Martinez Nielsen
Kristian Theilmann Gregersen

**Supervisor(s):**
Bent Thomsen
Thomas Bøgholm

**Copies:** 1

**Page Numbers:** 64

**Date of Completion:**
June 9, 2022

**Abstract:**

We explore the energy consumption of architectures of software applications. We investigate two architectures, namely the microservice and monolithic architecture. We implement the well described test-application, Pet Store in the languages C# and Java, following each architecture. We utilize JMeter, Docker and a wrapper implementation of Running Average Power Limit (RAPL) by Intel, to setup a test suite where controlled tests are run. Docker Compose and NGINX are used for scaling microservice testing. We setup a set of shell scripts to automatically conduct the tests. We use a test protocol and a minimal install of an operating system, for a consistent test setup. We find that the energy consumption per transaction in a non-scaling environment for C# is 43,79% to 38,80% higher in the microservice architecture compared to the monolithic architecture depending on the use of individual databases for each service. For Java it is 0,86% to 40,30% higher.

# Contents

# Preface

This project is a Master Thesis on the 4th semester of the Master of Software and the Master of Computer Science at Aalborg University. All code and implementations can be found on GitHub at `https://github.com/orgs/P10-energy-consumption/repositories` as well as `RAPL.rs` at `https://github.com/cs-21-pt-9-01/rapl.rs`. This project spans from 1st of February 2022 to 10th of June 2022. The project is supervised by Bent Thomsen[1] and Thomas Bøgholm[2].

Aalborg University, June 9, 2022

Daniél Garrido-Y Martinez Nielsen
<dgmn17@student.aau.dk>

Kristian Theilmann Gregersen
<kgrege16@student.aau.dk>

[1] `https://vbn.aau.dk/da/persons/110568`
[2] `https://vbn.aau.dk/da/persons/112525`

# Chapter 1

# Introduction

In the field of energy aware programming, studies have shown that coding style[5] [25], choice of programming language[4] and implementation specific details[23], all affect the energy consumption of a program. However, in order to fully develop a system that is energy aware, the software architecture should also be considered. The microservice architecture has already been widely adopted as a way to organize code due to its ability to scale and structure software, along with other advantages. While the microservice architecture is popular, it usually comes with different definitions. Some use it to organize code in smaller repositories, while others use it as a mean to easily scale applications. In this project, a microservice architecture is an architecture where an application is split into different services that each have access to its own database. However, structuring code as microservices, also increases the quantity of requests that an application produces, in order to communicate between the services. This likely also means an increase of energy consumption. However, to which order the energy consumption increases, is still an unknown. In this project, we investigate the energy consumption of microservice architecture, compared to a monolithic architecture.

This project is built on top of [22] and we will refer to this source throughout the project.

## 1.1 Motivation

The motivation behind this project arose from researching different architectures to compare. We studied an article that compared the resource allocation of microservices against monolithic architecture[16]. A graph from the article, quoted in Figure 1.1, says that it can be expected that microservices will outperform a monolithic architecture in terms of memory and that this trend only increases when the number of instances increases.

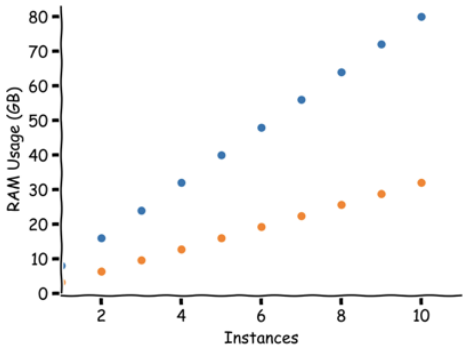



**Figure 1.1:** "Monoliths require more resources than Microservices, as more instances are running"[16]

In this project, we want to create a similar comparison. However, we are focusing on the energy consumption of the two architectures, when the number of instances increases.

## 1.2 Java Pet Store

A well described example that developers are familiar with is the Pet Store example. The Pet Store has its origins from May 2001 where Sun Microsystems presented the Pet Store solution implemented in Java. The goal of the solution was to showcase the best practices for developing J2EE™-based Web applications[20]. In November of 2001, Microsoft also implemented the Pet Store using their .NET environment. The purpose of the .NET implementation was to demonstrate how C# and .NET performed better than the Java implementation, and Microsoft also presented benchmarks results alongside the implementation. As the Pet Store is a well described example, the team providing the API documentation tool Swagger[1], also provides an implementation of the Pet Store, both for showcasing the documentation tool[2], as well as for their online editor[3]. In this project, we use the official Swagger documentation of the Pet Store, as a basis for our implementation of a monolithic- and microservice back-end. We searched GitHub for existing solutions and found the solution [4], from which we utilized the native SQL queries as well as the database creation script.

[1]https://swagger.io/
[2]https://petstore.swagger.io/
[3]https://editor.swagger.io/
[4]https://github.com/HenrikDK/PetStore

### 1.2.1 Microservice Candidates

Following the Pet Store example provided by Swagger, as well as a solution found at `https://github.com/HenrikDK/PetStore`, we extract three overall services: StoreService, PetService and UserService. We use these services as our base implementation with the possibility to extend each service with more functionality if needed. To facilitate communications between a hypothetical front end and the services, we also implement a GatewayService that functions as a middleware between the front- and back end.

## 1.3 Problem Hypotheses

To sum up the introduction chapter, we create a list of hypotheses based on the motivational section that we will study and try to confirm or deny in this project:

1. The monolithic architecture has a higher throughput / has a higher measurable performance, than the microservice architecture.
2. With no scaling, the energy consumption of the monolithic architecture, will be lower
3. Scaling the microservices will cause the microservice architecture to outperform the monolithic architecture in terms of power consumption, eventually.

# Chapter 2

# Technologies

In this chapter, we describe the different technologies that make up the tool-set that we utilize to conduct experiments. We base this chapter on the source [22]. For a detailed explanation for why different technologies were chosen, please refer to the source. For technologies that were considered in this project, we describe why they ultimately were chosen or discarded.

## 2.1 Automation of HTTP Requests

As we investigate the energy consumption of microservices, we will need a way of sending HTTP requests to the services and the monolith, containing the relevant data for the endpoints. To reduce human interference in the experiments, we considered different technologies and approaches to automate the process of sending requests.

### 2.1.1 Cypress

Cypress[1] is a JavaScript library for creating automated end-to-end tests written in JavaScript. The library supplies tools for asserting that specified HTML elements occur on a website. Moreover, Cypress also enables the monitoring of the relevant HTTP requests on a given website. For example, in a webshop, Cypress can monitor that the request for getting the product data, is being sent and it can assert on the HTTP response code. Cypress can also intentionally make sure that the request fails or is in a loading state, to assert that the correct loading elements are being shown. Though that Cypress provides a lot of relevant functionality, we decide on not utilizing it, due to a combination of not having a front-end service, but also for lacking functionality for our experiments.[6]

[1] https://www.cypress.io/

### 2.1.2 JMeter

JMeter is a Java application, created to do load testing on web-sites, but has since then expanded to be able to load test a multiload of services, while measuring the performance of the testing. Like Cypress, JMeter has functionality for sending HTTP requests. However, where Cypress does not provide functionality for scaling tests, JMeter provides functionality for specifying number of threads (users), to send the specified requests. Moreover, JMeter provides the ability to iteratively send these requests and the ability to specify for how long the tests should run, along with other relevant features such as a ramp-up period specifying how long JMeter should take to start a specified number of threads. Additionally, you are also able to define the request itself and you are capable of setting a request body or query parameters. As we want to be able to continuously send requests to our services, as well as to be able to control the parameters for how these requests are being sent, JMeter is chosen as the tool for testing our services.[15]

## 2.2 Deployment

Working with web applications, a form of deployment is needed. Simply deploying locally with a web server is a possibility but we look to find other possibly smarter options, as we want to deploy on a Linux computer in similar environments between the services. Moreover, we will need to be able to scale the microservices, which also emphasizes the importance of deployment technologies. Though, external services, applications and other most likely consume power themselves which should be considered during the collection of data.

### 2.2.1 Docker

Docker is an application allowing the ability to package software into containers that contain the needed libraries and tools to run the software [2]. Docker eases the deployment of applications along with the development and experiment of the implementations in relation to Operating System usage. Docker also allows for the C# and the Java version to run in similar environments. A tool for building the application and a `Dockerfile` describing the web server and a description of moving the built application to the web server is all that is needed for deploying in a Docker container.

This also ensures that nothing more than the web server running the application is running in the container and keeps the application isolated on the host machine. This allows for the possibility to isolate the performance of the container and thus the individual application.

Based on the ability to isolate the performance of each application, we choose to utilize Docker as a method of deployment.[7]

## 2.3 Scaling Microservices

One of the main motivations for using microservices is the ability to scale services such that whenever response times become increasingly high, a new instance of a service can be started on a new server. This way, the load may be distributed between the active instances to offload the services. This requires a way to organize and scale the individual services together with a load balancer to distribute load.

### 2.3.1 Kubernetes

Kubernetes is a Docker container orchestration tool. It allows for a detailed configuration of Docker containers that enables Kubernetes to facilitate a resilient and secure deployment environment. Kubernetes handles auto-scaling, what to do when services fail, load-balancing and more. Kubernetes serves a complex deployment environment, but requires a lot of setup. For this project Kubernetes seems excessive and requires a long setup period.[17]

### 2.3.2 Docker Compose

Docker Compose provides the ability to scale individual services by instantiating a specified amount of instances and automatically manage and assign them available ports. Simply referencing the exposed Docker container ports will grant access to the service. As a simpler alternative to Kubernetes, Docker Compose is ideal and chosen for managing the scaling as we use Docker for deployment.

### 2.3.3 NGINX

As we refrained from using Kubernetes, we need another load-balancer to distribute the requests to the services. Together with Docker Compose, NGINX may be used as a load-balancer by forwarding requests and will automatically reference the correct ports assigned in the Docker Compose setup. Given the ease of setup and coupling NGINX with the microservice application, it is used as a load-balancer for our applications.[21]

## 2.4 System Monitoring

While testing the services for energy consumption, we need to be able to monitor the energy consumption of the services. Moreover, it would be beneficial if we monitor the computer for other metrics such as memory consumption, CPU usage and network usage, as we will be more able to determine what causes the energy consumption. This is helpful to eliminate other factors such as the operating system doing background tasks, and cross referencing the test results.

### 2.4.1 Windows Performance Monitor

Windows Performance Monitor, or PerfMon, is a tool created by Microsoft to monitor the system. The tool enables one to create a range of custom data collectors, wherein you can specify metrics to monitor. For example, you can create a data collector, which monitors how many megabytes of memory are available. The tool also allows for specifying how often PerfMon should monitor the given metric and then log the output. Though PerfMon allows for detailed system monitoring, it will not be utilized in this project due to the need of utilizing a Linux operating system.[18]

### 2.4.2 Docker Stats

Docker Stats is a CLI command that one can provide to monitor the resource utilization of a Docker container. It is able to monitor memory consumption, CPU utilization, network recources and more. As we will deploy the services using Docker, Docker Stats is ideal for measuring the resource utilization of the different services.[8]

### 2.4.3 Running Average Power Limit

Running Average Power Limit or RAPL, is a tool created by Intel, to measure the energy consumption of the CPU, RAM and GPU in case of integrated graphics. It provides a detailed overview of different domains of the CPU and in turn allows for a detailed overview of where power is being consumed. For a detailed description of how RAPL functions, please visit chapter 3, in [22].[13]

### 2.4.4 Intel Power Gadget

Intel Power Gadget is a high level implementation of RAPL. It allows for monitoring the energy consumption of the CPU, RAM and GPU, as well as general usage. Intel Power Gadget provides the same functionality as RAPL and runs on the same platforms. However, you are restricted in the sense that you can not customize how often the system is being monitored and logged. Intel Power Gadget also provides the ability for starting the monitoring through a shell.[12]

### 2.4.5 RAPL.rs

`RAPL.rs` is an implementation of the RAPL tool provided by Intel, and is partly created by the authors of this project. The tool is a wrapper for RAPL written in Rust. Other than allowing the ability to run a specified application from within `RAPL.rs`, a logging session parallel to the running application is started to capture the energy consumption of the application. Thus, the RAPL logs provided by

`RAPL.rs` only contain information from the session of running the specified application skipping the otherwise necessity of filtering and processing the RAPL data. For a detailed description of the RAPL.rs tool, please visit section 4.2 in [22].

## 2.5 Summary

In summary, the technologies for setting up the test-setup comprises of a Docker setup, where the services will be running in containers in Docker. Docker enables us to easily scale and spawn instances of the microservices with Docker Compose and eases the deployment of the services to the Linux operating system. Together with Docker Compose, NGINX will be used as a load-balancer to distribute the requests when scaling the microservices. The operating system has to be a Linux operating system to be able to measure the energy consumption by using `RAPL.rs`. JMeter will be used to send requests to the services, which allows us to monitor statistics and be able to specify the amount of users, allowing for easy ramp-up of users to explore different loads. Finally, for monitoring the test computer, we utilize Docker Stats together with JMeter statistics, to monitor memory consumption, response time and CPU usage. For measuring the energy consumption, `RAPL.rs` will be utilized.

# Chapter 3

# Related work

In this chapter, a technique for isolating energy consumption within an application together with the state of the art related to tools and methodology for measuring performance of applications with microservice and monolithic architecture is presented.

## 3.1 The hunt for the guzzler: Architecture-based energy profiling using stubs

[14] presents a method for determining which parts of a software architecture draws the most power. They achieve this by studying the product application 'City Explorer', a dutch application that serves over 8000 annually. In the paper, they utilize JMeter together with bash scripts to automate testing, and they utilize Microsoft Joulemeter, as well as a WattsUp? Pro to determine the energy consumption. In the hunt of determining which parts of a software architecture consumes the most amount of power they present the sTEP method. The sTEP method comprises of different phases that make up the testing routine. In short, one have to select a functional component and describe what the component serves to do. Then you create an energy profile of the component by creating a test case for the component. Once you have a baseline energy profile, the next phase in sTEP method is to introduce stubbing, meaning that the component should do less computing. After stubbing the component, once again you create a description of the stubbed component and perform the same tests to create a new energy profile. The difference of power consumed between the stubbed element and the original, serves to provide information about the energy consumption of that exact element.

## 3.2 A Comparative Review of Microservices and Monolithic Architectures

[1] compare an implementation with monolithic architecture to one with a microservice architecture in relation to performance.

JHipster is utilized to generate a web application with a microservice architecture. JMeter is then used to test the performance in two test scenarios, namely load testing and concurrency testing. Lastly, a scenario testing the impact of different technologies is performed. The performance metrics captured are the throughput and response time.

JMeter is installed on a remote client and connected to the server through an Ethernet cable with Docker used to run the applications.

Load testing is performed by investigating the impact of increasing the user amount. 100 threads and a ramp-up of two minutes and a 'hold-time' of two minutes is utilized in the start where the number of threads is gradually increased until 7.000 threads. Results show that the throughput around 100 threads is significantly better on a monolithic architecture. The increase of threads brings the throughput closer for both architectures and results in an average difference of 0.87%. The amount of processed requests is also higher on a monolithic architecture when utilizing a lower amount of threads where the advantage is shifted towards the microservice architecture when the amount of threads is increased.

Concurrency testing is performed by sending 100 requests to each service with no ramp-up time and a gradually increasing amount of requests until 1.000 requests. This is to check the performance of the application with all services in use. The results show an average throughput increase for monolithic architecture compared to microservices of 6%. Though, no significant difference could be observed in relation to response time.

Lastly, 10.000 threads are used throughout 20 minutes of total runtime consisting of 10 minutes of ramp-up with five step ups increasing the number of threads by 2.000 until 10.000. After 10.000 threads are reached, the hold time is set to 10 minutes. JHipster utilizes a service discovery tool called Eureka but a new tool called Consul was added to the configurations in order to examine performance differences. Consul had a higher throughput of about 3.8% together with little better response time.

## 3.3 Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud

[28] also create two implementations of an application. One with monolithic architecture and another with microservice architecture. The implementations are

created with the Play web framework in Java.

JMeter is used to experiment with the performance. JMeter is configured to perform 30 requests per minute to one service, S1, and 1.100 requests per minute to another service, S2, for 10 minutes. Results show that the average response time for the monolithic architecture of S1 was lower than for the microservice architecture. Although, the microservice architecture had a slightly lower average response time for S2.

Microservices offer the advantage of being able to have small teams to maintain independent codebases using a more practical methodology in contrast to the monolithic architecture.

## 3.4 Performance evaluation in the migration process from a monolithic application to microservices

[11] examine the process of migrating from a monolithic application to microservice architecture with the help of models such as NGINX and IBM. An initial monolithic implementation is created using PHP programming language (1). Afterwards, the implementation programming language is changed to Java (2). Subsequently, services are identified and implemented using RESTful Web Services with design patterns (3) and without design patterns (4). Lastly, the microservices are fully implemented with frameworks and tools such as Spring Boot, Eureka and Zuul with design patterns (5) and without design patterns (6).

The performance of the different implementations is examined using JMeter. JMeter is configured to create 10, 100 and 1.000 threads simultaneously accessing 10.000 database records. Results show that failed executions are more frequent in implementations without design patterns. The CPU usage across all implementations are more or less the same. In regards to memory usage, the lowest memory consuming implementation is (1) but it should be noted that this implementation failed 100% of executions. The lowest memory consumption in relation to failed executions, is implementation (5). Implementation (3), (4), (5) and (6) utilized the network more than (1) and (2). This is to be expected as microservices utilize the network to make requests in contrast to regular function calls in monolithic implementations.

The performance in regards to the database is based on response time. The lowest response time in relation to failed executions, is implementation (5) followed by (3). It is concluded that performance-oriented patterns enhance the performance in regards to multiple aspects. Even more when they are combined with REST architecture.

## 3.5 From Monolithic Systems to Microservices: A Comparative Study of Performance

[27] examines a monolithic application running on a virtual server with KVM (Scenary 1) is compared to a microservice application running in Docker containers (Scenary 2) utilizing stress tests with the same hardware setups. A Non-Parametric regression model is applied to explain the dependency relationships between various performance variables.

Two JMeter cases are created. Case one consists of 273 requests, 10.000 generated data, 30 repetitions, three threads and three terminals. Case two consists of 1.053 requests, 20.000 generated data, 70 repetitions, five threads and three terminals.

Both cases are executed with JMeter GUI to view processed data. JMeter request results in Table 3.1 show that the microservice application is better at handling requests. Both in terms of throughput and overall speed for both cases while consuming more network data.

Furthermore, performance values in relation to hardware utilization and JMeter request results can be seen in Table 3.2 which show that the monolithic application utilizes the CPU less while utilizing disk and memory more. To no surprise, the microservice application utilizes the network more while still maintaining relatively low overall hardware utilization.

Overall, the microservice architecture outperforms the monolithic application in terms of efficiency. Though, as microservices communicate through network, slow and communication failure might occur making the architecture significantly more complicated in its nature.

| | **Case 1** | | **Case 2** | |
|---|---|---|---|---|
| | **Scenary 1** | **Scenary 2** | **Scenary 1** | **Scenary 2** |
| **Total requests** | | | 1053 | 1053 |
| OK | 273 | 273 | 1051 | 1053 |
| Error | 0 | 0 | 2 | 0 |
| **Duration time** | 00:01:50 | 00:01:29 | 00:14:17 | 00:12:08 |
| **Requests/s (average)** | 2.5/s | 3.1/s | 1.2/s | 1.4/s |
| **Duration per request** | | | | |
| Min | 7 ms | 4 ms | 22 ms | 8 ms |
| Max | 3677 ms | 2793 ms | 35,784 ms | 10,832 ms |
| Average | 1150 ms | 936 ms | 3934 ms | 3411 ms |
| Median | 695 ms | 312 ms | 2548 ms | 715 ms |
| Standard deviation | 1094.66 ms | 1082.4 ms | 38.05 ms | 4220.21 ms |
| **Total data** | | | | |
| Received | 85,014.43 KB/s | 105,193.29 KB/s | 86,342.29 KB/s | 102,433.12 KB/s |
| Sent | 0 KB/s | 0 KB/s | 0 KB/s | 0 KB/s |

**Table 3.1:** Results of the stress test of both scenarios from [27].

| Calculation | CPU Consumption (%) | Disk Reading (MB/s) | Disk Writing (MB/s) | Memory Consumption (MB) | Network Reception (B/s) | Network Transmission (B/s) |
|---|---|---|---|---|---|---|
| **Monolithic** | 2.3877 | 3.833604 | 12.40030 | 4.525155 | 3.285871 | 3.129852 |
| **Microservices** | 7.851149 | 3.833604 | 3.168329 | 2.887646 | 10.48274 | 4.505547 |

**Table 3.2:** Calculation of performance values from [27]

## 3.6 Container-based microservice architecture for cloud applications

[26] proposes a deployment methodology together with results from experiments on the performance differences between a monolithic and a microservice web application implementation. Additionally, scalability and deployment time is analyzed in relation to monolithic and microservice architecture.

JMeter is used to experiment with performance and two tests are performed. In the first test, JMeter created 50 threads with a ramp-up time of 10 seconds and a loop count of 1. With a low amount of threads, the difference in response time is negligible while the performance for microservice architecture compared to the monolithic architecture gradually increases along with the amount of threads suggesting better performance when more load is applied.

In the second test JMeter created 2.000 threads with a ramp-up time of 100 seconds and a loop count of 10. Results show that the microservice architecture outperforms the monolithic implementation. The load is distributed between instances of services running in the microservice implementation while the monolithic implementation need the whole application to process a single request. As with the first test, the microservice architecture performance increases gradually compared to the monolithic architecture in relation to throughput.

In regards to scalability and deployment time, the microservice architecture is favored due to only needing to apply changes to individual services in contrast to the whole application.

# Chapter 4

# Methodology

We present the methodology that we follow in order to create a consistent and reliable test setup that in turn enables us to gather clean data. We describe how we implement our test systems and what technologies we use to test and measure the energy consumption of said systems. We describe in depth the test setup and the testing protocol we use.

## 4.1 Implementation

Following the microservice candidates we defined in 1.2.1, we implement each service. The structure of the different implementations are described in Figure 4.2 and Figure 4.3 with their respective endpoints described in Table 4.1, 4.2 and 4.3.

As the Pet Store was originally developed in C# and Java, we also decide on this. We create two implementations in the two languages to examine whether performance tendencies can be seen or if we are able to find performance discrepancies between the implementations. This also enables us to more accurately determine how the architecture itself contributes to the energy consumption. For `POST`- and `PUT`-requests, the services receive data formatted in JSON (JavaScript Object Notation) as parameters, while in `GET`- and `DELETE`-requests, we follow RESTful standards and parse the parameters in query parameters through the URL path. Examples of each type can be seen in Table 4.1 with `{id}` denoting the path parameter and `{status}` denoting the query parameter.

| HTTP Method | URL Path | Description |
|---|---|---|
| **GET** | /pet/{id} | Fetches a pet with the specific ID in the store |
| **POST** | /pet | Adds a pet to the store |
| **PUT** | /pet | Update a pet in the store |
| **DELETE** | /pet/{id} | Deletes a pet with the specific ID in the store |
| **GET** | /pet/findByStatus?status={status} | Fetches all pets with the specific status |

**Table 4.1:** Pet service endpoints.

| HTTP Method | URL Path | Description |
|---|---|---|
| **GET** | /user/{username} | Fetches a user with the specific Username |
| **POST** | /user | Adds a user |
| **PUT** | /user | Update a user |
| **DELETE** | /user/{username} | Deletes a user with the specific Username |

**Table 4.2:** User service endpoints.

| HTTP Method | URL Path | Description |
|---|---|---|
| **GET** | /store/inventory | Fetches amount of pets for each status |
| **GET** | /store/order/{id} | Fetches an order with the specific ID |
| **POST** | /store/order | Adds an order |
| **DELETE** | /store/order/{id} | Deletes an order with the specific ID |

**Table 4.3:** Store service endpoints.

For the database, we utilize the database implementation that was found in the solution described in section 1.2. We use this design in order to save time. Though database selection may affect the energy consumption, this is not in the research-scope of this project. The PostgreSQL database consists of four tables. Namely, `order`, `pet` and `user` with a variety of columns of different types to vary the processed data in the services. The database structure can be seen in Figure 4.1.

### 4.1.1 Monolithic Architecture

Figure 4.2, depicts the architecture of the monolithic Pet Store. We expose three endpoints, which all access their own respective repository. The repositories live in the data access layer, which has a direct connection to a PostgreSQL database.

| pet | |
|---|---|
| id | integer |
| name | character varying |
| category | integer |
| status | integer |
| tags | character varying |
| created | timestamp |
| createdby | character varying |
| modified | timestamp |
| modifiedby | character varying |
| deleted | timestamp |
| deletedby | character varying |
| isdelete | boolean |

| order | |
|---|---|
| id | integer |
| petid | integer |
| quantity | integer |
| shipdate | timestamp |
| status | integer |
| complete | boolean |
| created | timestamp |
| createdby | character varying |
| modified | timestamp |
| modifiedby | character varying |
| deleted | timestamp |
| deletedby | character varying |
| isdelete | boolean |

| user | |
|---|---|
| id | integer |
| username | character varying |
| firstname | character varying |
| lastname | character varying |
| email | character varying |
| passwordhash | character varying |
| salt | character varying |
| phone | character varying |
| status | integer |
| created | timestamp |
| createdby | character varying |
| modified | timestamp |
| modifiedby | character varying |
| deleted | timestamp |
| deletedby | character varying |
| isdelete | boolean |

**Figure 4.1:** PostgreSQL database structure.

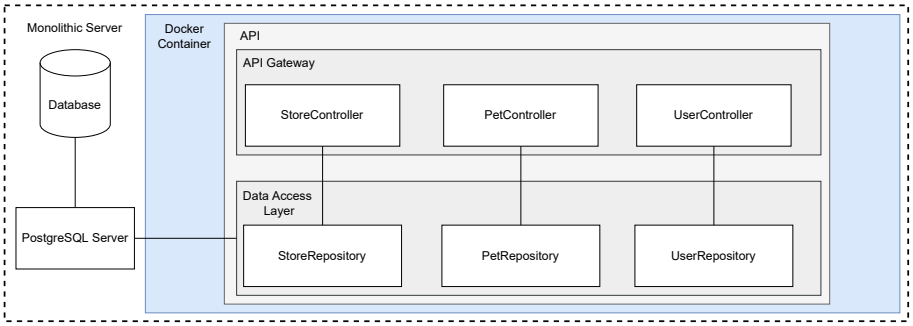



**Figure 4.2:** Monolithic Architecture

The controllers expose the endpoints and is where the requests will hit the service. Often a layer containing the business logic will exist between the controllers and the data access layer. However, extra logic is not necessary for the our testing.

### 4.1.2 Microservice Architecture

Figure 4.3 depicts the architecture of the Pet Store, organized in microservices. The architecture consists of an API gateway, which exposes the same endpoints as the monolithic solution. However, whereas the monolith directly accesses the datalayer, the API-gateway distributes the requests to the respective service.

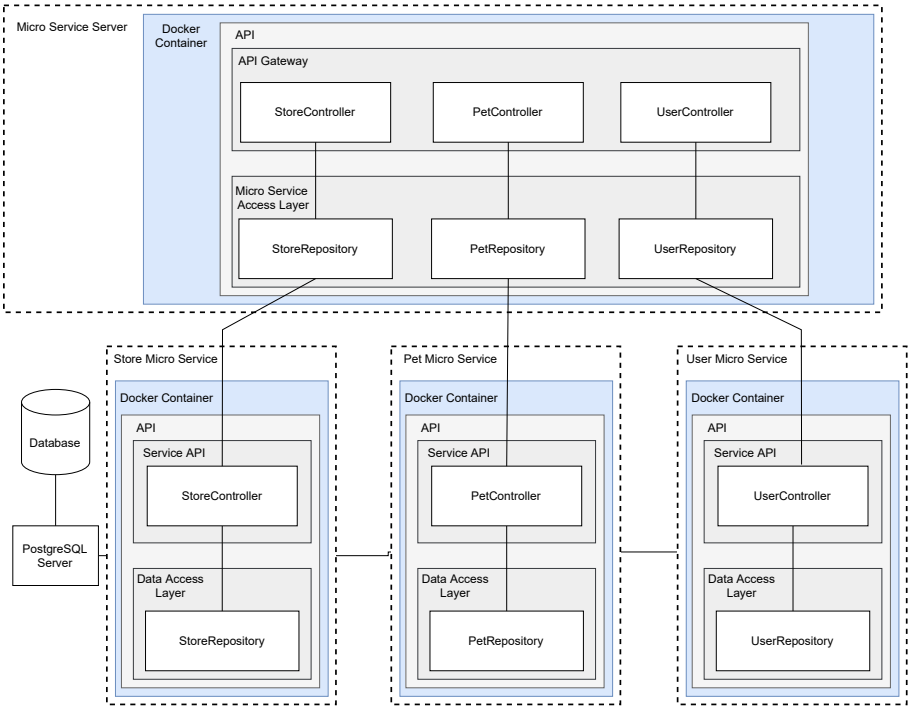



**Figure 4.3:** Microservice Architecture

Each service, exposes the same functionality as the repositories in the monolithic solution. Likewise, in each service, the repositories live in the data access layer and each have a direct connection to the PostgreSQL database.

### 4.1.3 .NET

The .NET implementation of the architecture, takes it starting point in the ASP.NET Core Web API template. The template contains an example controller that implements a RESTful service. Moreover, the template also contains a Docker container. The .NET version implements the architecture described in subsection 4.1.1 and 4.1.2. Dependency injection is setup to manage the repositories and the specific implementation is thus injected into controllers at runtime. The default resource lifecycle is RequestScoped which means a new instance of a resource is created with each new request in contrast to a singleton with only one instance per application.

The .NET implementations are written in .NET 6.

### 4.1.4 Java

Following the same architecture described in subsection 4.1.1 and 4.1.2, two Java implementations are created with their respective architecture. `Jakarta RESTful Web Services` (JAX-RS)[1] is utilized to create service resources along with `WildFly 25.0.0.Final`[2] for deployment of the services.

For the monolithic implementation, an incoming request handled by an endpoint in a Controller is converted to a POJO (Plain Old Java Object) with the exception of endpoints accepting Path and Query parameters. The POJOs are then converted into regular Java objects with types such as dates. This is done for easy object and data manipulation and interaction. The data received by the endpoint is further sent to Repositories to be included in native SQL queries. The queries are simply executed through an open database connection and the result is returned back to the Controller. Lastly, a response of the result is built and returned to the client.

With the case of the microservice implementation, as described in subsection 4.1.2, the API-gateway is required to send the request further to the responsible microservice Controller. In order to achieve this, the raw request received is attached to a new regular HTTP request and sent to the respective service. The individual services are then responsible for serializing the result for the response. Lastly, the HTTP responses are deserialized when needed for the API-gateway response. Other than that, the functionality remains the same as with the monolithic implementation.

Likewise, as with the .NET implementation, the JAX-RS default resource lifecycle is RequestScoped.

## 4.2 JMeter

As a part of automation and minimizing human interference, JMeter 5.4.3 is utilized to automatically send requests to our services[3]. Which tests we want to conduct and the JMeter configurations to execute the tests are described in this section.

### 4.2.1 API Testing

To fully explore the performance of the implementations under different loads, one would preferably configure a set of test plans that each test a specific attribute. Services often undergo extensive testing to make sure that the service in question

[1] `https://docs.oracle.com/javaee/7/api/javax/ws/rs/package-summary.html`
[2] `https://www.wildfly.org/news/2021/10/05/WildFly25-Final-Released/`
[3] `https://jmeter.apache.org/usermanual/component_reference.html`

is able to handle the expected amount of users. Below is a set of test that a service might undergo before reaching production.

- Spike testing (∼30% CPU utilization with Random 100% CPU utilization)
- Linear Load Ramp-Up (Linearly increase thread count)
- Stepping Load Ramp-Up (Gradually increase thread count)

However, in this project we are not interested in testing if the services are capable of handling different levels of load, but rather interested in the energy consumption across the different levels of load. Because of this, we setup the JMeter test plan to follow a Stepping Load Ramp-Up test methodology. To run a Stepping Load Ramp-Up test, we set up a set of requests in JMeter which each utilize every endpoint of the services. To control the load and increasingly ramp-up the load, we configure JMeter to control the number of threads for a given time period. How this test is conducted in described in subsection 4.2.4.

### 4.2.2 JMeter Thread Groups

As a way to structure your JMeter test plan, JMeter provides thread groups. A thread group is a controller for threads determining the number of threads, how long JMeter should spend starting the threads and how many times a test should be repeated. This means JMeter also gathers what requests the threads should be making.

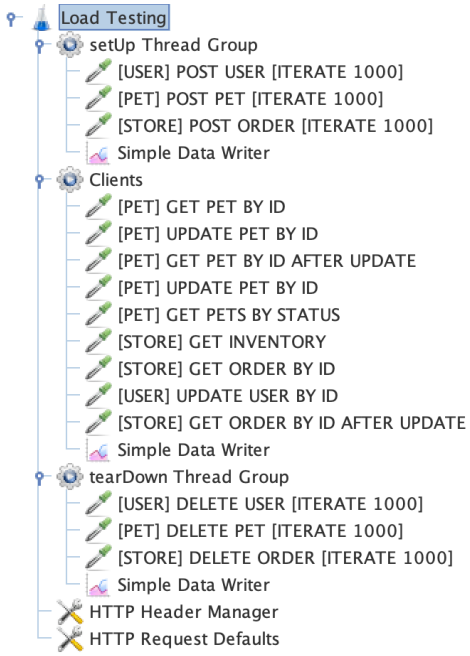



**Figure 4.4:** JMeter Thread Groups and Requests

Whenever a thread is spawned in a thread group, the test (collection of the requests) in the group is executed in order. In our test plan, we utilize three thread groups: a setup thread group, a testing thread group and a tear-down thread group. Figure 4.4 depicts the requests that make up the test plan. The figure also depicts the three thread groups.

The `setUp` thread group is responsible for setting up the database content and prepare for testing by making 100 post requests to each service while `tearDown` is responsible for cleaning up after testing leaving an empty database. The `Clients` thread group is responsible for the actual testing by making requests to endpoints retrieving and manipulating data.

### 4.2.3 JMeter Test Plan

The test plan described in section 3.2 from [1] uses the same form of thread count ramp-up by gradually increasing the amount of threads running concurrently. The thread count increases from 100 to 7.000 with a 'hold-time' of two minutes.

The test plans used in [26] described in section 3.6 uses a form of ramp-up as well. Though, in their case, it is done linearly from zero to 50 threads and another test from zero to 2.000.

Similar to the studies described in section 3.3, 3.4 and 3.5, we perform a test to measure and record the response time and throughput of the applications in an attempt to compare performance of monolithic and microservice applications.

By researching similar papers mentioned in chapter 3, we aim to create a test plan which takes inspiration from the related work, but also fits the testing needs in this project.

### 4.2.4 Stepping Load Ramp-Up

JMeter is extended with a plugin called *Ultimate Thread Group*[4] used to create thread groups allowing for a ramp-up in thread count by more than one unlike the standard ramp-up period simply allowing a linear thread count increase. This thread group is used in conjunction with the `Clients` thread group to steadily increase load after a specific delay. Every seven and a half minutes, the thread count is increased in order to record performance of the implementation in a single test. In a seven and a half minute test run, no anomalies in response times were uncovered, justifying the length of the test for a given thread count.

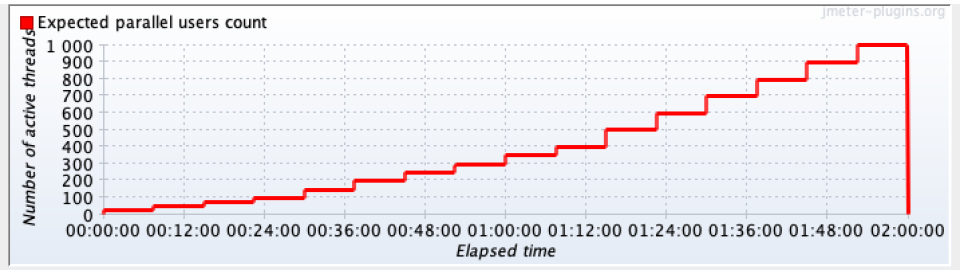



**Figure 4.5:** Clients thread group configuration - visualized

The total number of threads, together with how the thread count is being ramped up, is depicted in Figure 4.5. The exact configuration can be seen in Table A.1 and A.2 in the appendix. The increase amount is doubled at thread count 100 and 400 to speed up the test plan. The response time and energy consumption is recorded and examined afterwards which means we are able to uncover whether a significant event occurs in between each thread count increment.

This specific test plan is used for all testing.

## 4.3 Experiments

An experiment with a singular shared database server is to be conducted as this is a common occurrence for monolithic applications.

Furthermore, a test giving each service their own database running in Docker containers to manage is conducted exploring how this would impact the response time and energy consumption. Having a database instance with only data needed

[4]https://jmeter-plugins.org/wiki/UltimateThreadGroup/

for that service is needed to stay true to the concept of microservices in relation scalability and the management of individual services [9].

The summary, the following experiments will be conducted:

- Singular shared database server
- Individual database server

## 4.4 Measurement Protocol

To further decrease the error margin and human interference and increase validity, we setup a measurement protocol. The measurement protocol consists of a set of shell scripts that automatically execute the test plan that we are to run. In total we create three shell scripts: `TestSystemBuildAndKillDocker.sh` is responsible of building and starting the relevant Docker containers and lastly kill the Docker processes when the test is over. `TestSystemJMeterRunTestPlan.sh` simply starts with a `sleep` command followed up by the execution of a JMeter test plan and ending with a `sleep` command. The `sleep` commands are used to gather idle data used for comparing how much energy the individual applications consume. Lastly, `TestSystemRunAllBenchmarks.sh` handles all the timing of starting the `TestSystemBuildAndKillDocker.sh` process, starting process of gathering hardware utilization metrics and starting `RAPL.rs` with the `TestSystemJMeterRunTestPlan.sh` shell script.

To further ensure validity for each test, we define a measurement protocol that executes each of the shell scripts. The measurement protocol also ensures that we follow a set of steps to make each as equal as possible.

1. Restart system
2. Run test script
   (a) Build and run services
   (b) Start logging Docker hardware utilization
   (c) Execute JMeter test plan
   (d) Stop services and logging
   (e) Repeat step (a), (b), (c) and (d) for next application

`RAPL.rs` is responsible for executing the JMeter test plan and ensures that energy consumption from RAPL is logged to a file. JMeter creates log files containing results in relation to throughput, response time, etc. from the execution of a test plan while a background Docker Stats process logs hardware utilization metrics.

## 4.5 Test Setup

A single test setup was used consisting of a single test system in order to ensure consistency and validity in results obtained from testing. The system and the tests are conducted in the same location as to reduce the impact of external factors and limit the factors to the individual implementations and their respective performance.

The test system hardware specifications can be seen in Table 4.4 and the software specification in Table 4.5.

| **Hardware** | **Specification** |
|---|---|
| **Computer type** | Dell OptiPlex 5050 (07A2) |
| **Motherboard** | Dell 0WWJRX |
| **RAM** | 2x 8GB; 2400 MHz; DDR4; DIMM;<br>Micron Technology (8ATF1G64AZ- 2G3B1) and<br>Hynix Semiconductor (HMA81GU6AFR8N-UH) |
| **CPU** | Intel i7-6700; Skylake; 3.4GHz<br>(overclock: 4.2GHz); 64bit; clock 100MHz |
| **GPU** | Integrated |
| **Disk** | 256GB SSD; INTEL SSDSC2FK25 |

**Table 4.4:** Hardware specification of test system.

| **Software** | **Specification** |
|---|---|
| **Operating system** | Ubuntu Server 21.10, minimal install |
| **Login manager** | lightDM |
| **Window manager** | i3 (X11) |
| **C#** | .NET 6.0 |
| **Java** | OpenJDK 11.0.12+7-Ubuntu-0ubuntu3 |
| **PostgreSQL** | PostgreSQL 14 |

**Table 4.5:** Software specification of test system.

We made sure to install Ubuntu Server 21.10 as a minimal install, to reduce energy consumed by background processes of the operating system. A common practice in a measurement protocol is to close unrelated processes running on the operating system. However, as we have made sure to install this specific operating system, almost no unnecessary processes are running. Moreover, we have not installed any other programs than what has been necessary to run the tests.

# Chapter 5

# Results

In this chapter we describe the results from running the test plans described in subsection 4.2.4 using the measurement protocol mentioned in section 4.4. However, in chapter 6 we will discuss the findings and explain the various results.

## 5.1 Expectations

Before presenting the results, we will share what we expect from the testing. After having studied the article[16] mentioned in section 1.1, we expect the monolith to perform better in terms of throughput, response time while facilitating fewer total requests. However, as depicted in Figure 1.1, we expect that the more instances that is instantiated, the lower the resource utilization becomes for the microservice architecture, when compared to the monolithic architecture. From this, we also expect for the energy consumption to follow a similar trend, which is also why we are ramping up the number of threads increasingly as described in subsection 4.2.1. Following subsection 2.3.2, which describes how we can scale the microservices with Docker, we also expect that at some threshold the throughput would eventually be higher with the scaled microservices than in the monolith. However, we did not form any expectations for when this would happen or what the energy consumption would look like. However, it is important to research this threshold and its consequences in terms of throughput and energy consumption, as otherwise, microservices could just be scaled infinitely without describing the downsides to the architecture.

## 5.2 Testing Restrictions

A large part of the microservice architecture is the fact that one can scale the services when needed. Upon running the experiments on the test computer, we dis-

covered a range of restrictions when scaling the microservices. In subsection 2.3.2, we describe how we utilized Docker Compose in order to orchestrate the Docker containers to run multiple instances of each service to achieve this simulation of scaling. However, upon running the test plan on the scaled microservices, we found that the throughput was ultimately lower on the scaled microservices, compared to a non-scaled environment. We concluded that this is due to the fact that we are limited in computation power with the current JMeter test plan and that spinning up more instances on a single computer only leads to a lower throughput due to more request having to be sent through the network, while running at a 100% CPU load.

| **Implementation** | **C# Microservice Preliminary** | **C# Microservice Scaling Preliminary** |
|---|---|---|
| **Total transactions** | 7.889.433 | 5.764.767 |
| **Transactions/s** | 1.947,97 | 1.423,20 |

**Table 5.1:** Preliminary throughput results for C# microservices and microservices with scaling with individual databases.

Ideally, multiple servers on multiple computers would be setup, but this would require a complete restructuring of the testing setup. Moreover, when comparing energy consumption of two architectures, we would like that the implementations ultimately reflect each other as much as possible. With scaling microservices, one would have to ask the question of when to stop the scaling and how many instances one should run, in order to create a fair comparison. Without determining these parameters, microservices could be scaled infinitely, leading to an infinite energy consumption in theory. Because of this, all tests and results reflected in this chapter, only concerns with microservices vs. monolithic architecture, in a non-scaling environment, where we did not use the load-balancer, nor NGINX.

## 5.3 C# Single Shared Database

This section describes the results for the microservice and monolithic architectures for the C# implementation, utilizing a single shared database. The section describes the results for all metrics gathered during the execution of the JMeter test plan.

### 5.3.1 Energy Consumption

The energy consumption results from the respective architectures are depicted in Figure 5.1 and Figure 5.2. The figures show the energy consumption delta values, which is the difference between the previous recorded energy consumption, and

the current. Thus, we are able to see possible increases or decreases in energy consumption in relation to the JMeter test plan.

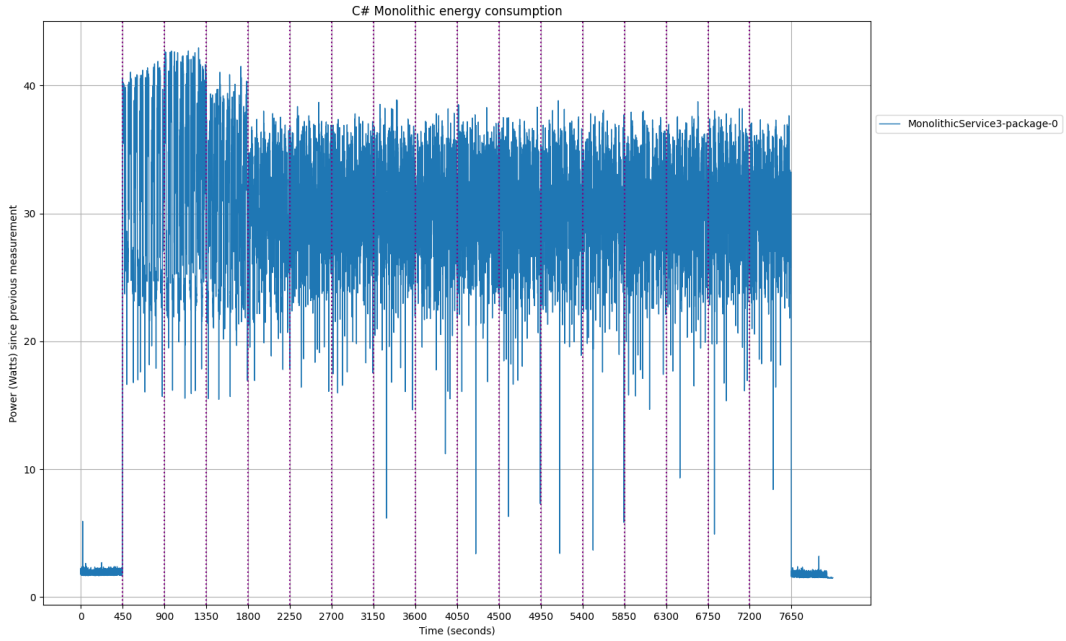



**Figure 5.1:** Energy consumption from C# monolithic service with a single shared database.

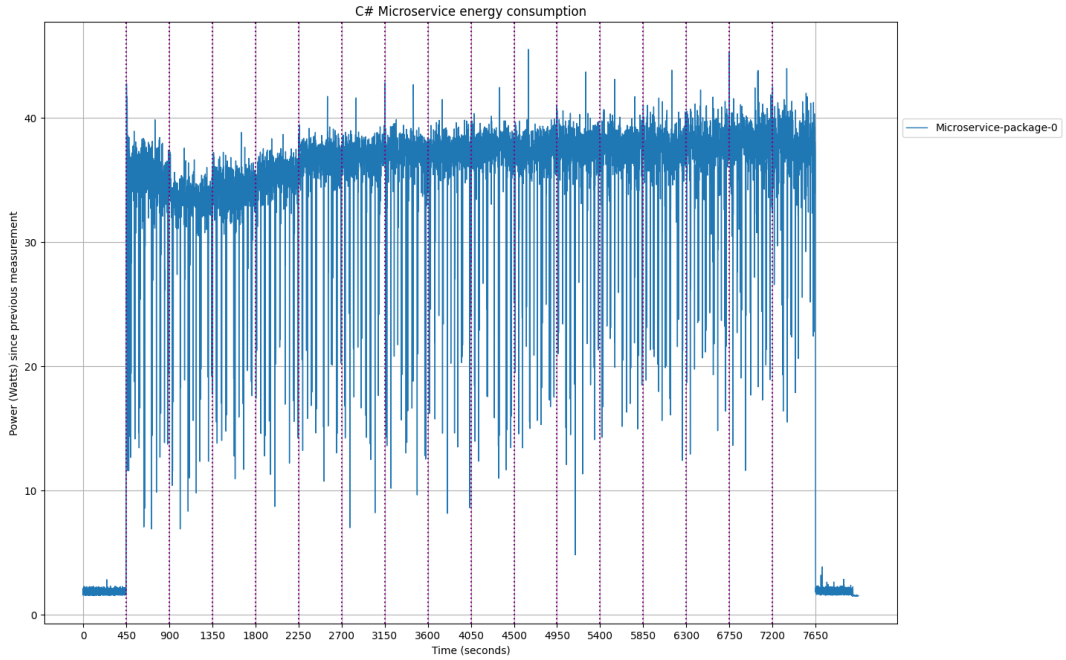



**Figure 5.2:** Energy consumption from C# microservice with a single shared database.

In the figures it is possible to see a lack of energy consumption in the first and last 450 seconds. This is a part of the test plan where the internal sleep command has been called to showcase the baseline energy consumption of the system, before any test is running. After the initial 450 seconds, we see the energy consumption of the running JMeter test plan. Every 450 seconds is marked by a purple dotted vertical line in order to mark the increase in thread count to assist in showcasing the effect. In both Figure 5.1 and Figure 5.2, we see that initially the energy consumption is high during the beginning of the test plan execution. However, at 1.800 second mark, going from 75 to 100 thread count, the energy consumption decreases immediately leading into a consistent consumption while still showcasing similar fluctuations as before the decrease.

### 5.3.2 CPU Utilization

The CPU utilization for the monolithic and microservice implementations can be seen in Figure 5.3 and Figure 5.4, respectively. Due to the test system consisting of a processor with multiple cores, the CPU percentages reach above 100% in relation to the utilization of cores.

For the monolithic architecture, the CPU usage is just around 0% during the initial and ending sleep command as expected. This increases drastically increases to about 330% for the 450 seconds after the sleep command with a thread count of 25. The CPU usage slightly decreases over the next thread count increments down to about a steady 240% usage up until the ending sleep. For the microservice architecture, we see that the CPU usage of the individual services is remarkably lower. However, if you add up each service the CPU usage would be higher in total.

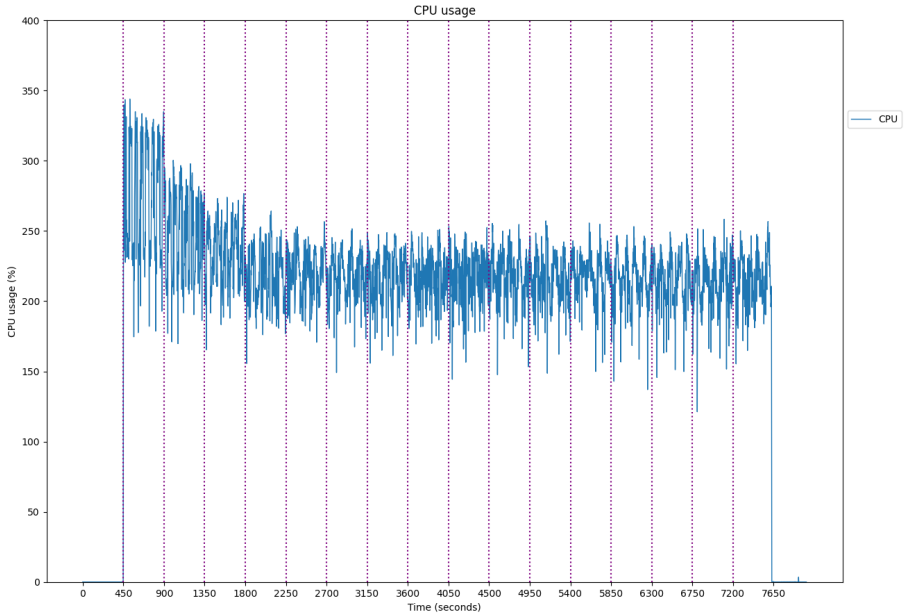



**Figure 5.3:** CPU usage results from C# monolithic service with a single shared database.

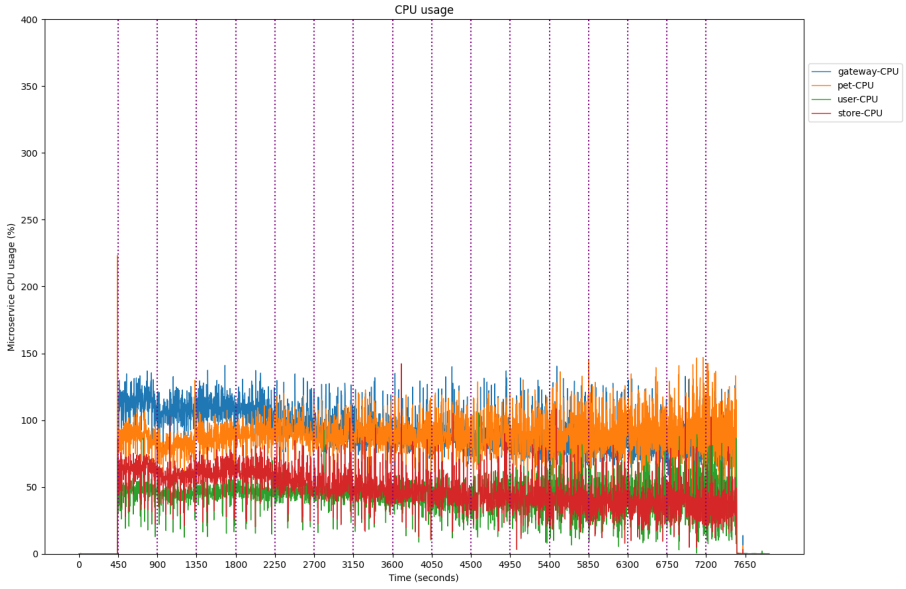



**Figure 5.4:** CPU usage results from C# microservice with a single shared database.

### 5.3.3 Response Time

The response time results can be seen in Figure 5.5 and Figure 5.6 for the monolithic and microservice implementations, respectively. The data is grouped in values in 1 minute intervals for better presentation. The different colored plotted lines each showcases the response time for a given request in the JMeter test plan.

In Figure 5.5 the ramping up of threads described in subsection 4.2.4 is represented. There is a clear resemblance between Figure 5.5 and Figure 4.5. Likewise, in the microservice architecture, a similar trend is seen, however, not as clearly as with the monolithic architecture.

From examining the CPU utilization, the response time results are likely due to the CPU being under 100% load with the application constantly attempting to serve all requests. As more requests are waiting to be served by the increase in thread count, the response time increases explaining the steady ramp-up in response time. This problem is caused by the nature of our JMeter setup.

As the response times do not provide any valuable information about the application, we choose to examine the throughput as a measure of performance.

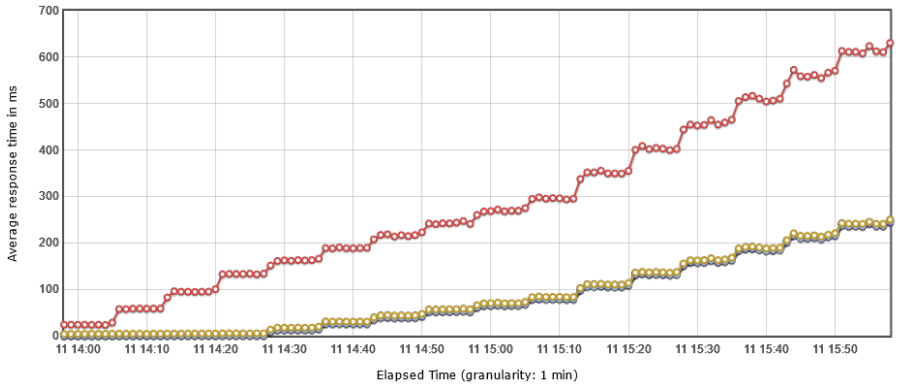



**Figure 5.5:** Response times with grouped values in 1 minute intervals from C# monolithic service with a single shared database.

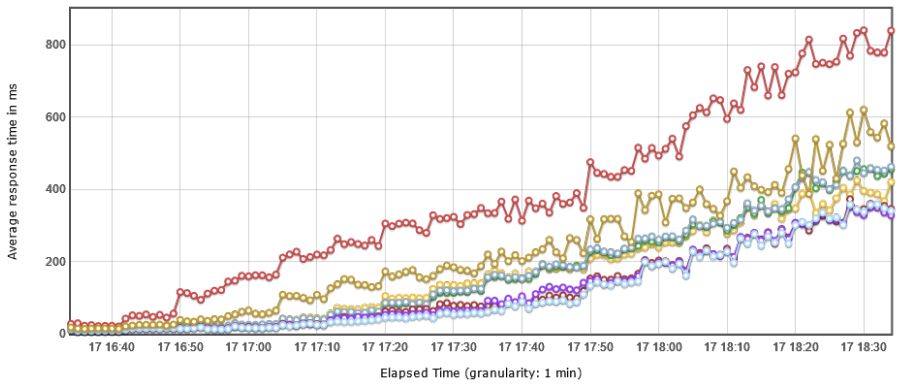



**Figure 5.6:** Response times with grouped values in 1 minute intervals from C# microservice with a single shared database.

### 5.3.4 Throughput

The throughput (transactions per second) is considerably higher for the monolithic application by about 39,27% over the microservice application.

| Implementation | C# Monolith | C# Microservice |
|---|---|---|
| **Total transactions** | 22.822.178 | 13.859.193 |
| **Transactions/s** | 3.169,61 | 1.924,79 |

**Table 5.2:** Throughput results for C# monolithic service and microservice with a single shared database.

## 5.4 C# Individual Microservice Database

This section describes the results for the C# implementation with an individual database for each service from the execution of the JMeter test plan.

### 5.4.1 Energy Consumption

Examining the results from giving the microservices each their own database in Figure 5.7, the small decrease in energy consumption seen early on for the implementation without individual databases is not present for the counter implementation. This results in a slightly higher average energy consumption.

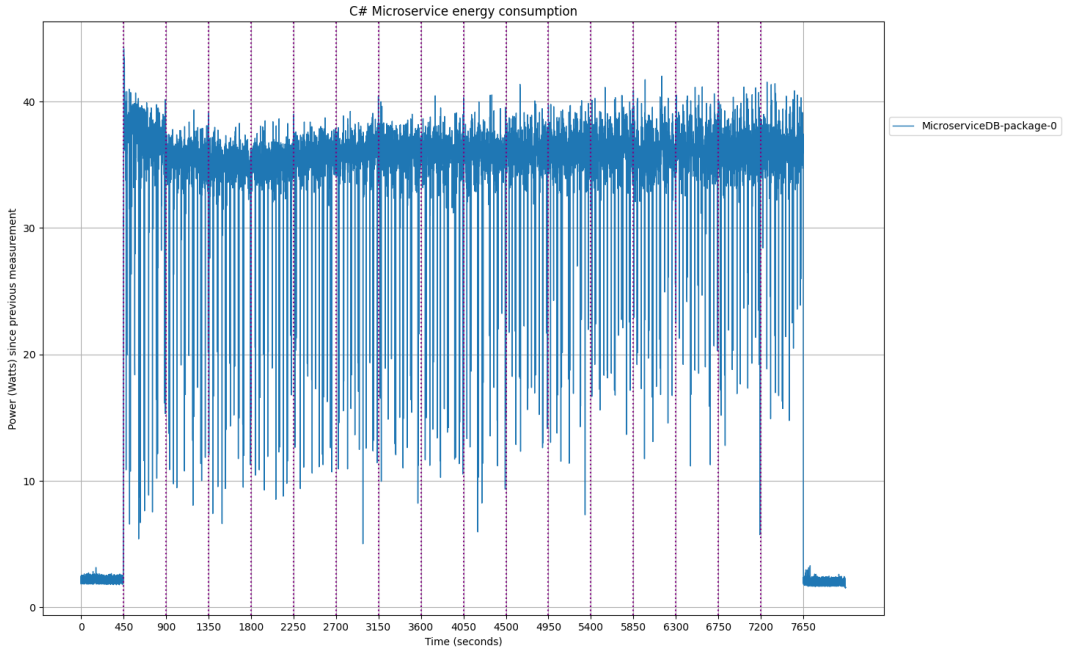



**Figure 5.7:** Energy consumption from C# microservice with individual databases.

### 5.4.2 CPU Utilization

The CPU utilization for C# microservice with individual databases can be seen in Figure 5.8 and shows very similar results to CPU usage without individual databases as seen in Figure 5.4.

The CPU usage is slightly lower due to the databases consuming some of the CPU processing power. Otherwise, the CPU usage is maxed out giving the same pattern of response times as previously.

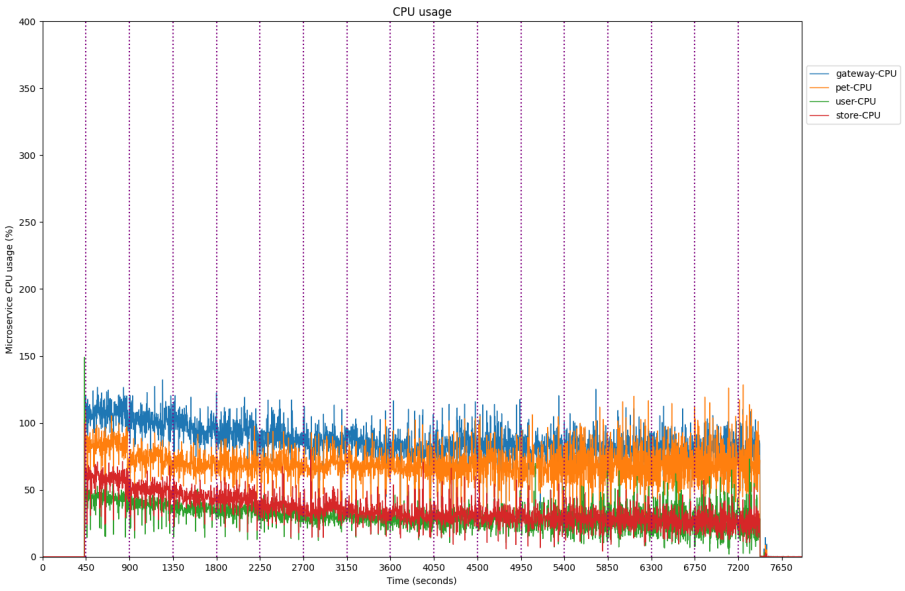



**Figure 5.8:** CPU usage results from C# microservice with individual databases.

### 5.4.3 Throughput

This experiment provided the microservice with individual databases with a higher throughput than with a singular database by about 6,42% while still being lower than the monolith.

| Implementation | C# Microservice DB | C# Microservice |
|---|---|---|
| **Total transactions** | 14.810.071 | 13.859.193 |
| **Transactions/s** | 2.056,88 | 1.924,79 |

**Table 5.3:** Throughput results for C# microservice with individual databases.

## 5.5 Java Single Shared Database

This section describes the results from all the metrics that were recorded during the JMeter test plan execution of the Java implementations.

### 5.5.1 Energy Consumption

The energy consumption results from the microservice and monolith test can be seen in Figure 5.10 and 5.9, respectively. The sleep duration is reflected on the graph from zero to 450 seconds showing low energy consumption as expected during a sleep. Afterwards, the JMeter test plan is executed and causes a large increase in energy consumption from 450 to about 900 seconds and slowly fades

into a stable consumption of about 51 Watts to lastly end the JMeter execution resulting in a low and stable consumption during the ending sleep to end the test.

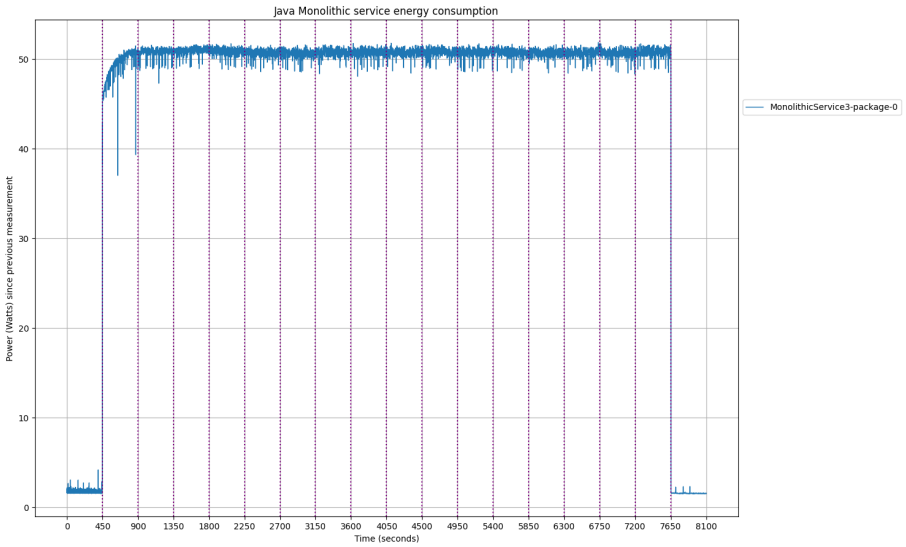



**Figure 5.9:** Energy consumption results from Java monolithic service with a single shared database.

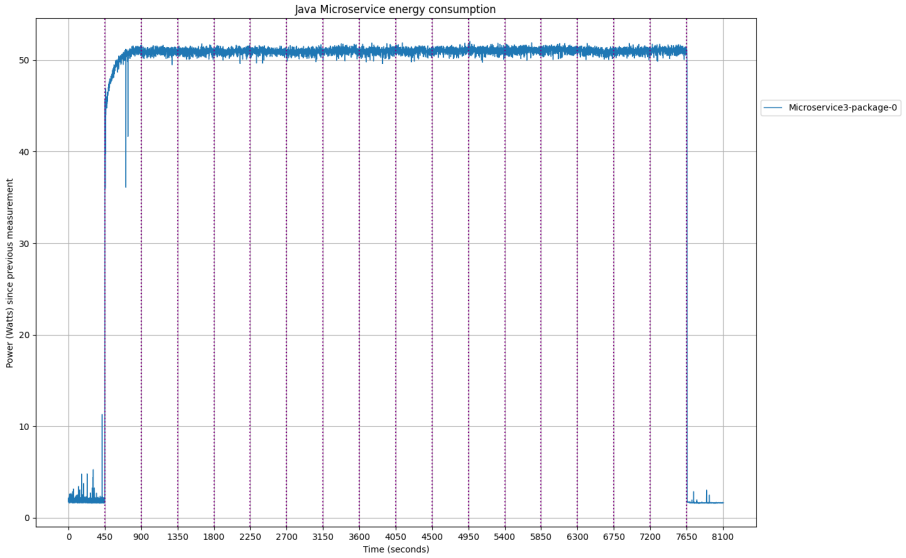



**Figure 5.10:** Energy consumption results from Java microservice with a single shared database.

### 5.5.2 CPU Utilization

The CPU utilization for the Java monolithic and microservice implementations can be seen in Figure 5.11 and Figure 5.12, respectively.

The monolithic service has about a 560% CPU utilization quickly decreasing to a steady utilization of about 540%.

Somewhat similar for the microservice, the CPU utilization steadies quickly after a small decrease. Furthermore, all services except the gateway starts fluctuating more than previously during the test around 4950 seconds with a thread count of 600.

Summing the individual CPU utilization values for the microservice shows about the same CPU utilization as the monolithic service simply spread between services.

Given that the CPU usage is not increasing along with the increase in thread count, we can assume that the Java applications are under maximum load and that the response time simply increases steadily due to more requests from more threads similar to the C# applications.

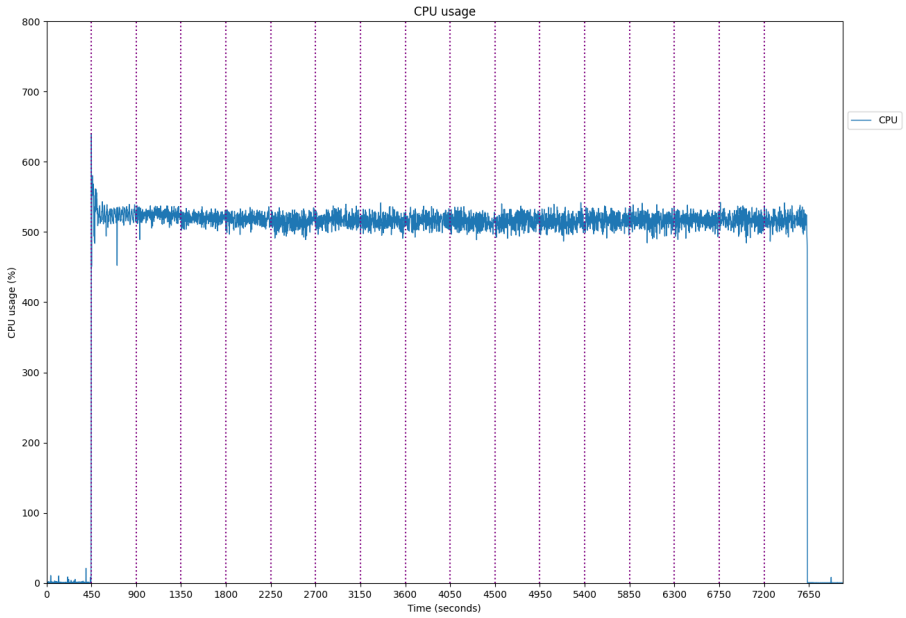



**Figure 5.11:** CPU usage results from Java monolithic service with a single shared database.

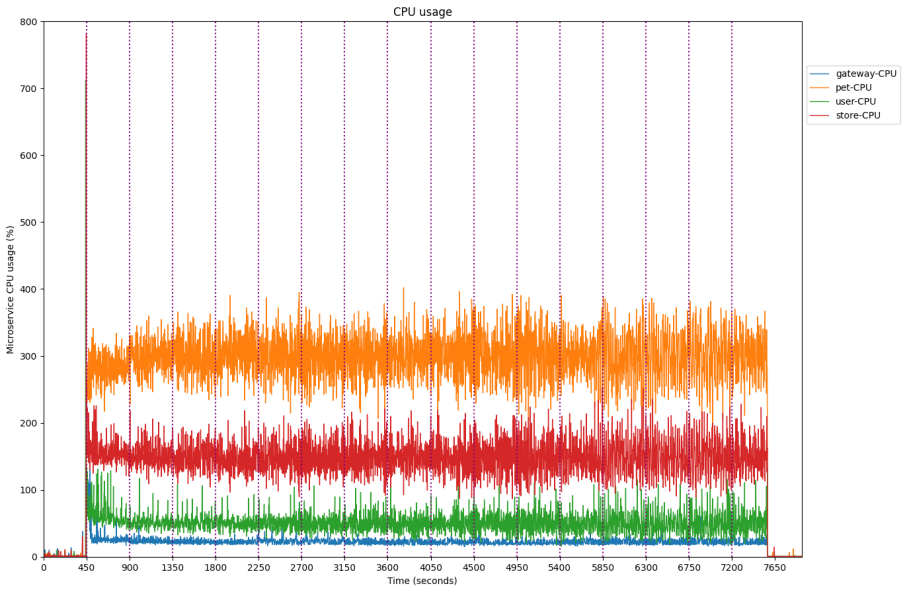



**Figure 5.12:** CPU usage results from Java microservice with a single shared database.

### 5.5.3 Throughput

The monolith and microservice implementations achieve almost identical throughput results with the monolithic implementation reaching a throughput 0,29% higher.

| **Implementation** | **Java Monolith** | **Java Microservice** |
|---|---|---|
| **Total transactions** | 2.130.473 | 2.124.268 |
| **Transactions/s** | 295,76 | 294,90 |

**Table 5.4:** Throughput results for Java monolithic service and microservice with a single shared database.

## 5.6 Java Individual Microservice Database

This section describes the results for the Java implementation with an individual database for each service from the execution of the JMeter test plan.

### 5.6.1 Energy Consumption

The energy consumption can be seen in Figure 5.13 and is very similar to the experiment with a singular shared database except for the fluctuations between 51 and 45 watts.

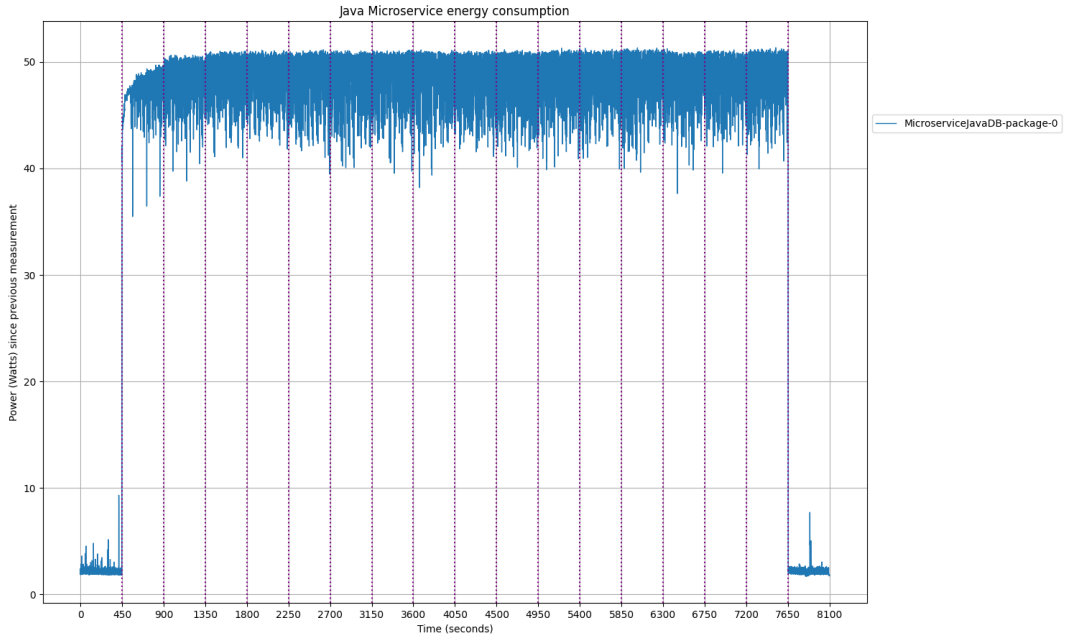



**Figure 5.13:** Energy consumption results from Java microservice with individual databases.

### 5.6.2 CPU Utilization

The CPU utilization is very similar to the experiment with a singular shared database while seemingly having smaller fluctuations.

As with the C# experiment, the CPU usage in this experiment is slightly lower due to the databases consuming some of the CPU processing power as well. Otherwise, the CPU usage is maxed out giving the same pattern of response times as previously.

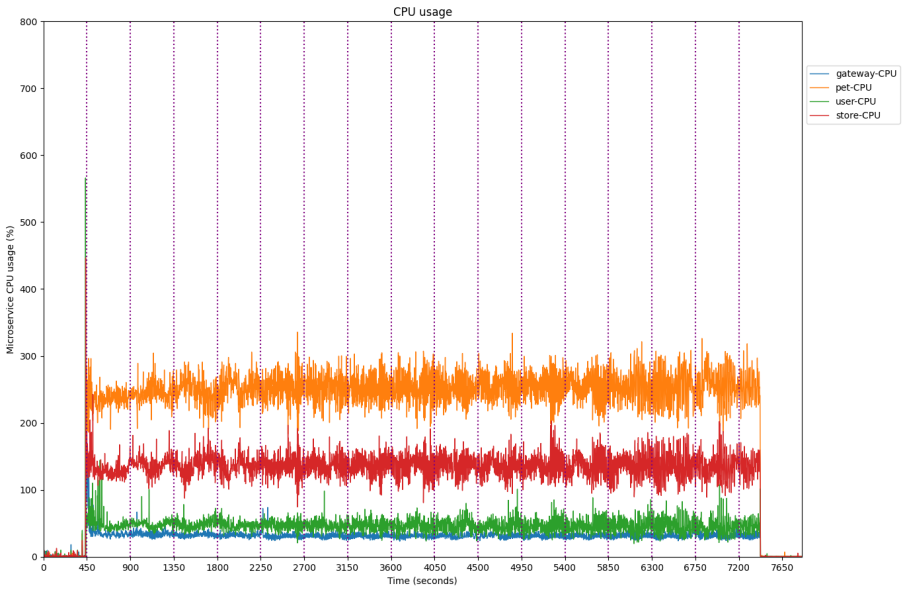



**Figure 5.14:** CPU usage results from Java microservice with individual databases.

### 5.6.3 Throughput

The throughput is significantly higher for this experiment boasting an increase of 37,73% which is also significantly higher than the monolith.

| **Implementation** | **Java Microservice DB** | **Java Microservice** |
|---|---|---|
| **Total transactions** | 3.410.771 | 2.124.268 |
| **Transactions/s** | 473,59 | 294,90 |

**Table 5.5:** Throughput results for Java microservice with individual databases.

# Chapter 6

# Discussion

In this chapter we discuss the findings described in chapter 5. We try to explain outlier datapoints and why the energy consumption, CPU utilization, response time and throughput looks the way it does. Moreover, we discuss the threats to the validity of this project as well.

## 6.1 C# Single Shared Database

This section relates to the results described in section 5.3. The purpose of this section is to try to explain the various results.

### 6.1.1 Energy Consumption

An overlap of the energy consumption results from the C# implementation can be seen in Figure B.1 in the appendix and shows that the microservice implementation generally consumes more energy. Simultaneously, the microservice implementation has a stable consumption in contrast to the monolithic implementation which consumes less but fluctuates more. This observation is supported by the computed standard deviation and mean seen in Table 6.1 which is computed from start to end of the JMeter test plan execution to focus on the actual usage of the application.

The difference in total energy consumption by each implementation is not large considering the length of the experiment. Though, when examining the total amount of transactions processed, the monolith implementation achieves significantly better results. These values allow us to compute an energy pr. transaction result which is averaged over all JMeter requests.

| Implementation | C# Monolith | C# Microservice |
|---|---|---|
| **Standard deviation** | 4,88 | 4,44 |
| **Mean** | 30,50 | 35,75 |
| **Total power (J)** | 240.056,71 | 259.354,70 |
| **Energy pr. transaction** | 0,011 | 0,019 |

**Table 6.1:** Standard and mean energy consumption together with RAPL statistics for C# monolithic service and microservice with single shared database.

The C# microservice implementation uses about 43,79% more energy pr. transaction which is further emphasized considering energy consumption when the sample count is evened. If we were to consider a sample count of 10.000.000 transactions, the energy consumption for the monolith was to reach $10.000.000 * 0,011 = 110.000,00J$ while the microservice was to reach $10.000.000 * 0,019 = 190.000,00J$ which is a difference of $80.000J$ possibly realized in two hours from any application that constantly serves clients under full load.

The microservice implementation might use more power due to using more memory. Even though the instances are smaller than the monolith, significantly many more instances to process requests are made all while double the amount of controllers are created to handle redirecting requests to the respective services.

This makes choosing between the two architectures a significant decision in relation to both performance and energy consumption considering the use of C#.

### 6.1.2 CPU Utilization

As described in subsection 5.3.3, the CPU is likely used 100% constantly. Interestingly, the monolith CPU usage is decreasing as the thread count increases. This is likely due to the fact that JMeter CPU utilization increases as it is required to send more requests to the application. Otherwise, the utilization is to be expected considering the testing restrictions described in section 5.2.

### 6.1.3 Response Time

The monolith implementation generally reaches lower response times than the microservice variation which additionally, fluctuates more. This is likely because of the application being split into services. There might not be as many pet service instances as store instances while their will be monolith instances according to the amount of JMeter request.

However, as described in subsection 5.3.3, as more requests are waiting to be served by the increase in thread count, the response time increases together with a CPU usage of 100% explaining the steady ramp-up in response time. This issue

renders the response time results unusable and we instead look towards throughput.

### 6.1.4 Throughput

The network communication overhead from utilizing microservices is very apparent in the throughput results. The implementations both utilize instance-per-request making the only difference the network communication between services and memory usage. With low response times, the importance of being able to quickly respond with services close to each other is highlighted through throughput. With a monolithic application, the services do not get any closer and thus achieves superior results.

## 6.2 C# Individual Microservice Database

This section relates to the results described in section 5.4. Similar to previous section, the purpose of this section is to try to explain the various results.

### 6.2.1 Energy consumption

Due to the C# implementation with individual databases not having the small decrease in energy consumption in the start, a slightly higher mean energy consumption and lower standard deviation is reached as seen in Table 6.2. This is likely due to the increase in database instances running and is to be expected as no other additions are made.

As the throughput is about 6,42% higher together with an overall slightly lower energy consumption, the microservice test with individual databases for each service achieves about 8,16% lower energy pr. transaction but about 38,80% higher energy pr. transaction than the monolithic implementation. This gives this implementation the advantage in terms of both energy consumption and throughput.

| **Implementation** | **C# Microservice DB** | **C# Microservice** |
|---|---|---|
| **Standard deviation** | 4,52 | 5,91 |
| **Mean** | 35,06 | 31,32 |
| **Total power (J)** | 254.541,15 | 259.354,70 |
| **Energy pr. transaction** | 0,017 | 0,019 |

**Table 6.2:** Standard and mean energy consumption for C# microservice with single shared database and microservices with individual databases.

Similar to the CPU utilization for the test with a single shared database, the CPU usages show no noticeable differences.

### 6.2.2 Throughput

The throughput is slightly higher for the microservice with individual databases (2.056 requests per second to 1.924 requests per second) likely due to the database instances not being under as much load as the single instance for the other test and allowing for better performance.

## 6.3 C# Energy Consumption Ramp-up

Both in Figure 5.1 and Figure 5.2, we initially see an increase of energy which soon after reduces to a stable level. After having studied the ASP.NET environment and its documentation, we find an article that describes the lifecycle of an ASP.NET application.[19] We find that in ASP.NET projects, an HttpApplicationManager class is being instantiated upon an incoming request to the application. However, the documentation also states that due to the garbage collector of .NET, the same HttpApplicationManager can be cached and reused to improve performance. We initially suspected that the initial increase of energy consumption was due to a process like the one in ASP.NET projects where .NET instantiates new objects, until relevant objects have been cached. However, since the following energy consumption flattens and appears consistent, it seems that the API is under 100% load from shortly after the initial beginning of the test. This could also explain the rise and then immediate decrease in energy consumption, because we initially see an increasing load on the API, corresponding to the energy increase, until the API is under 100% load, leading to a steady energy consumption. This also explains why we do not see a further increase in energy consumption, while the thread count is increasing.

### 6.3.1 Energy Fluctuation

Another trend in the energy consumption is that the microservice architecture seems to fluctuate a bit more than the monolithic architecture. After having studied the diagnostic tool that Visual Studio provides, we suspect that the fluctuation occurs due to the services are awaiting for either receiving a request or awaiting the API-gateway's processing of the response. The diagnostics tool shows us that a service utilizes very little CPU usage when idling. Since this is the case, whenever a service awaits for a request, the service would almost be at idle energy consumption, which could explain the fluctuation.

## 6.4 Java Single Shared Database

This section relates to the results described in section 5.5. Similar to section 6.1, the purpose of this section is to try to explain the various results.

### 6.4.1 Energy Consumption

A comparison of the two implementations in relation to energy consumption can be seen in Figure B.2 in the appendix. Overall, the results comparing the two architectures show little to no difference in energy consumption during the test. The results show the same tendencies which is also showcased by the standard deviation and mean energy consumption that can be seen in Table 6.3. The effect of choosing between the two architectures for an application and usage scenario comparable to the one modelled in the JMeter test plan used in this project is negligible (0,86%).

This allows developers to focus on practical aspects when choosing between these architectures. This is quite surprising given that the microservice needs to create a connection to a respective services and parse the result from it leading to believe that the microservice architecture would use more processing power and thus consume more energy.

| **Implementation** | **Java Monolith** | **Java Microservice** |
|---|---|---|
| **Standard deviation** | 0,94 | 0,95 |
| **Mean** | 50,62 | 50,89 |
| **Total power (J)** | 368.038,42 | 370.185,62 |
| **Energy pr. transaction** | 0,17 | 0,17 |

**Table 6.3:** Standard and mean energy consumption together with RAPL statistics for Java monolithic service and microservice with single shared database.

### 6.4.2 CPU Utilization

Interestingly, the overall CPU usage percentage is significantly higher than the C# counterpart. About 500% for the Java implementations and 200 to 250% for the C# implementations which we were unable to find a reason for.

### 6.4.3 Throughput

Similar to the overall energy consumption of each Java implementation, the throughput shows no significant differences. This is interesting given the same reasoning as for the energy consumption.

## 6.5 Java Individual Microservice Database

This section relates to the results described in section 5.6. Similar to section 6.1, the purpose of this section is to try to explain the various results.

### 6.5.1 Energy Consumption

The similarity between the energy consumption from the Java experiment with and without individual databases for each service is very apparent. The only difference is the fluctuation between the steady 51 Watts seen without individual databases and 45 Watts. Similar to the experiment with C#, this is likely due to the increase in database instances running and is to be expected as no other additions are made. This is quite a significant advantage as this leads to an overall lower energy pr. transactions as seen in Table 6.4. An advantage of about 40,30% to be specific. The overall energy consumption is not far from each other but the throughput is instead mostly responsible for the advantage.

| **Implementation** | **Java Microservice DB** | **Java Microservice** |
|---|---|---|
| **Standard deviation** | 2,56 | 0,95 |
| **Mean** | 48,50 | 50,89 |
| **Total power (J)** | 351.742,43 | 370.185,62 |
| **Energy pr. transaction** | 0,10 | 0,17 |

**Table 6.4:** Standard and mean energy consumption together with RAPL statistics for Java microservice with single shared database and microservice with individual databases.

Very similar to the CPU utilization for the Java test with a single shared database, the CPU usages show no noticeable differences.

### 6.5.2 Throughput

The throughput is significantly higher for the microservice with individual databases (473,59 requests per second to 294,90 requests per second) and is again likely due to the database instances not being under as much load as the single instance for the other test and allowing for better performance. This is similar to the experiment with C# except Java benefits significantly more from this change.

## 6.6 Java Energy Consumption Ramp-up

A version of Figure B.2 without the initial sleep and stable energy consumption is depicted in Figure 6.1 showcasing the slow increase in energy consumption. This is a contrast to other studies showcasing energy consumption in relation to start-up

and warm-up of the Java Virtual Machine (JVM) usually showing a slow decrease in energy consumption as these phases are surpassed.

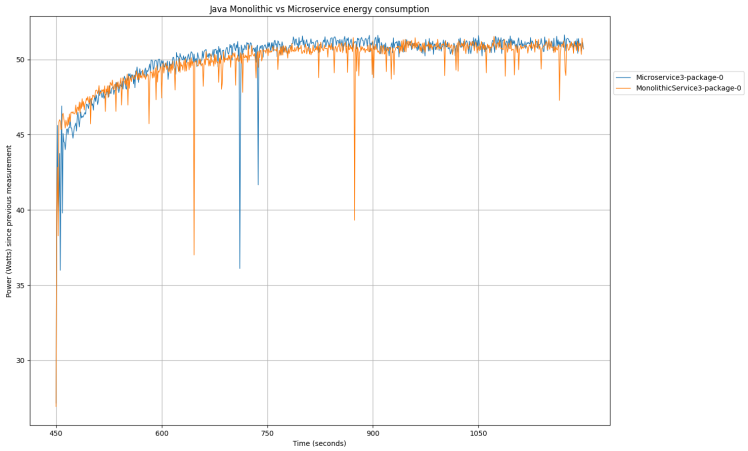



**Figure 6.1:** Energy consumption results from Java microservice and monolithic service from 450 to 1.200 seconds with single shared database.

Looking at the logs, it does not seem to be a phenomenon stemming from JMeter as all threads are up and running in a matter of a few seconds while the phenomenon is happening for about 250 seconds. This leads us to believe that it either relates to the Just-In-Time (JIT) compiler, the fact that the application is under heavy load, an increasing amount of processes and memory usage or a combination.

The progression of memory usage and process count can be seen in Figure 6.2 and 6.3, respectively. As indicated by the memory usage, the phenomenon would be explained if it were not for the memory usage waiting to increase until about 1.800 seconds and not at the start of the execution. The amount of processes helps visualize the increase in thread count and how it related to how the application handles the requests by creating several instances. Though, it does not indicate a relation to the energy consumption as the amount of processes at the time of increase in energy consumption is relatively stable and low.

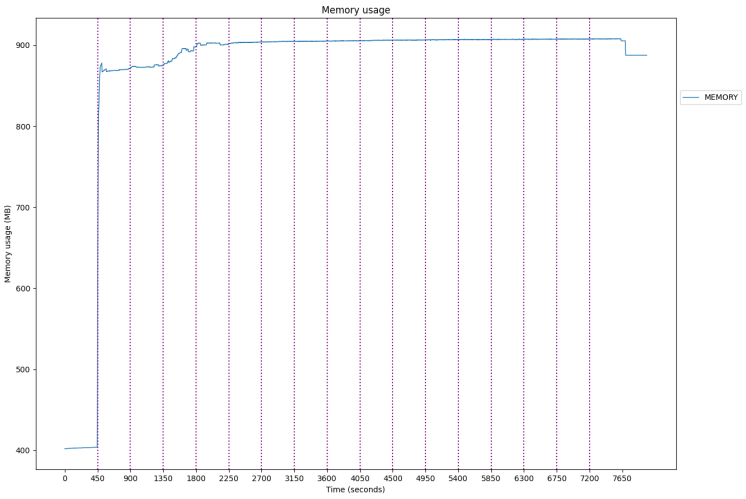



**Figure 6.2:** Memory usage results from Java monolithic service with single shared database.

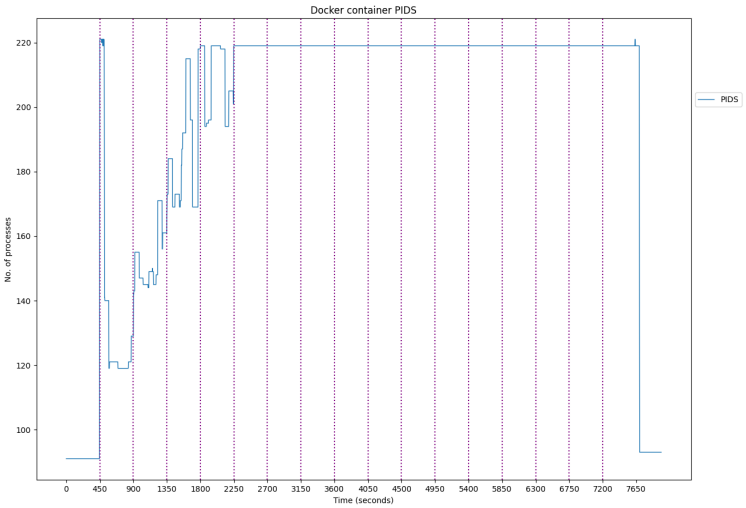



**Figure 6.3:** Amount of processes running in Docker container progression results from Java monolithic service with single shared database.

The JIT compiler might not have compiled the whole application at the time of starting the JMeter test plan and may done on the fly whenever needed and whenever code is executed the first time, the compiler spends energy compiling. Though, it would seem that all the different JMeter requests are processed well before 250 seconds have passed making it unlikely to be the sole explanation. Zakaria Ournani et al.[24] explores energy consumption of different JVMs and showcase that the energy consumption almost immediately increases to the peak consumption and, depending on the service, decrease or maintain a stable consumption for

all tested distributions. Even HotSpot which is utilized by the OpenJDK version installed on our test system. This is in contrast to what we experience.

## 6.7 Threats to Validity

This section discusses the threats of the validity of this project and we will provide recommendations for what could be done to increase the validity.

### 6.7.1 Local Requests

A part that we consider a threat to the validity of this study, is the fact that we only tested on a single system, sending requests locally. Energy consumption from a client, as well as a server would be needed in order to create a better picture of the cost of running a microservice architecture, when it comes to energy consumption. There also is the energy consumption cost of communicating on the network, which also contributes to the final energy consumption. Coroama et al.[3] tries to provide a formula for the energy consumption of the internet. The study of Coroama et al. could also be incorporated in determining the true energy consumption of microservices that are hosted.

### 6.7.2 Business Logic

The business logic in this study have been reduced to the minimal to reduce external factors like implementation differences between Java and C#. However, in real world applications business logic might be complicated and communicate between a number of microservices. This could introduce loading and a higher load in each service. This is not the case for this study, where each service does not communicate with one another, and the business logic is minimal. To increase validity, testing on real world applications could be an interesting contribution.

## 6.8 Future Test Proposal

To lower the threats to validity, described in section 6.7, we want to share our thoughts and ideas for a future setup that improves on the areas which threatens the validity of the findings. In a future iteration, this testing setup could be used for future experiments.

### 6.8.1 Test Setup Proposal #1

Figure 6.4, depicts an ideal testing setup. The figure shows that the client and the servers are separated. Moreover, it shows that additional computers are needed for

each service, especially whenever scaling the microservices is needed, as to reflect the real world. Finally, in the figure, we also see that the energy consumption will be measured across the client, API-gateway and the microservice servers.

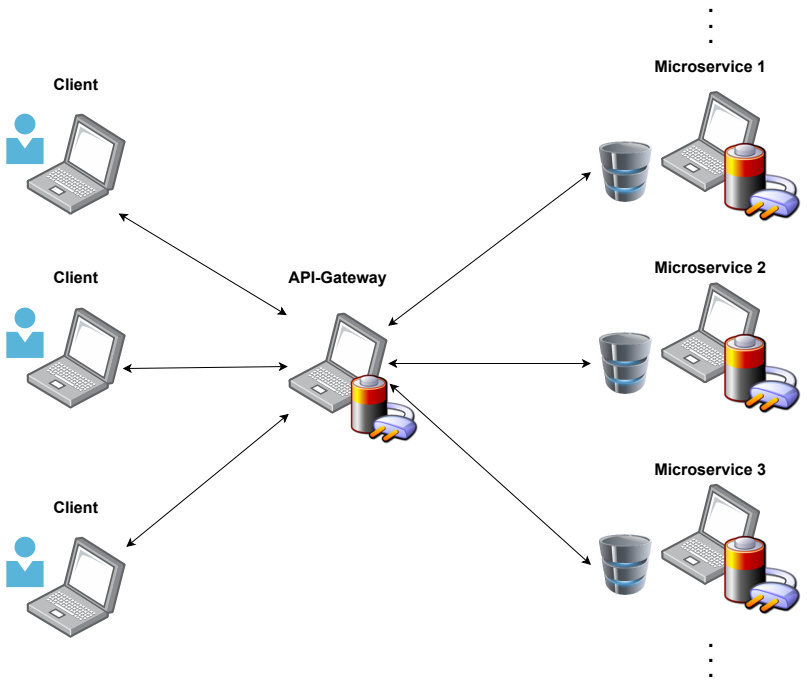



**Figure 6.4:** Test Setup Proposal #1

Besides the physical testing setup, we recommend that for the testing software in the test setup, `RAPL.RS`, JMeter, Docker and NGINX can be used like it has been used in this project. Together with bash-scripts, everything can be deployed automatically and execution can be done from a bash-script, reducing human interference. We do however, recommend that the JMeter test plan should be modified to fit the needs of a given new experiment. With additional computation power with the services living on their own server, ramping up like has been done in this project might be possible, without maxing out the testing setup. Likewise, a testing setup where one is not ramping up and might be mimicking real world web applications, could also be utilized.

### 6.8.2 Test Setup Proposal #2

For the second ideal test setup, the major difference is the extraction of the client on its own computer. This way the true energy consumption of the architecture is what is being measured, and not JMeter as well. Moreover, we propose that the testing plan should be modified to not utilize 100% of the CPU at all times.

By controlling how much the CPU is being utilized, scaling the services virtually with Docker on a single computer can be done successfully. However, we still deem the fact that all the services are limited and being run on a single server, is not completely reflective of the real world. However, this setup would lower the threats to the validity.

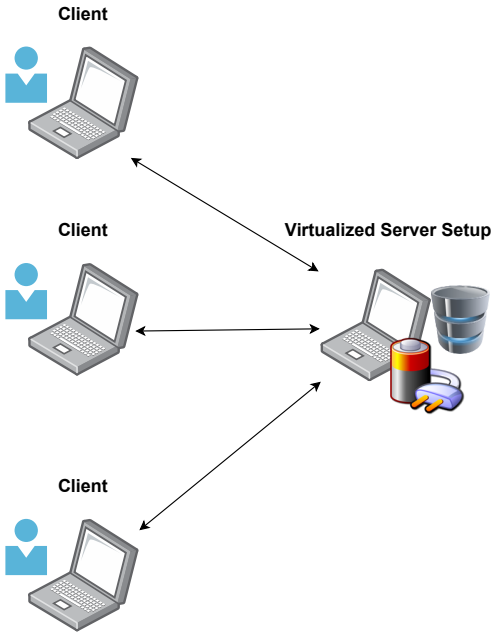



**Figure 6.5:** Test Setup Proposal #2

Figure 6.5 depicts the testing setup in this project, however, one where the clients have been extracted. In this setup, all the microservices and the API-gateway would all live on the virtualized server setup. In this setup, it is important to control the CPU utilization of the services with JMeter, such that is it possible to scale the microservices on the setup.

# Chapter 7

# Conclusion

This chapter concludes the project and sums up the findings. It also has a section on ideas for future iterations of the project.

## 7.1 Conclusion

In this project we have studied the energy consumption of the microservice- and monolithic architecture. We have created two implementations of the Pet Store for examination, and ramped up the number of users for each solution to determine if scaling affects the energy consumption, similar to how it affects the resource usage as described in section 1.1.

Due to the fact that we are limited in computation power while scaling the microservices, the experiments in this study can only relate to the comparison of microservices against monolithic architecture, in a non-scaling environment. This comparison is still relevant as many utilize microservices to structure code in smaller repositories and to reduce downtime, thus the findings are still relevant for determining the architecture of a real world application. When scaling is not considered we can compare the results, but will ultimately be an unfair comparison due to the fact that a large part of microservices is in fact the ability to scale.

Our findings show that in a non-scaling environment, the energy consumption per transaction for C# is 43,79% to 38,80% higher in the microservice architecture compared to the monolithic architecture depending on the use of individual databases. For Java it is 0,86% to 40,30% higher. Using individual databases improves performance but does not necessarily improve energy consumption which means using individual databases for each service may greatly improve the resulting energy consumption due to throughput. We also find that the throughput is higher in a monolithic architecture, which is in line with the expectations.

However, we should also conclude that this does not mean that microservices are inferior in energy consumption. One could still speculate that in certain edge

cases when scaling is implemented, microservices would be more energy efficient. Lastly, peculiar ramp-up in energy consumption can be seen for Java application likely related to the utilized JVM which should be further experimented with.

### 7.1.1 Hypotheses

To sum up the results, we refer back to the hypotheses listed in section 1.3. We want to answer the following:

1. The monolithic architecture has a higher throughput / has a higher measurable performance, than the microservice architecture.
2. With no scaling, the energy consumption of the monolithic architecture, will be lower
3. Scaling the microservices will cause the microservice architecture to outperform the monolithic architecture in terms of power consumption, eventually.

For the first hypothesis (1), we can confirm the hypothesis - in a non-scaling environment, as we find that the monolithic architecture, in both implementations, has a higher throughput. For the second hypothesis (2), we can also confirm the hypothesis, as we find that in a non-scaling environment, the monolithic architecture outperformed the microservice architecture. For the third hypothesis (3), we can not confirm or deny the hypothesis, as the testing for the scaling microservices, were restricted by the testing setup.

## 7.2 Future Work

In this section we discuss what future iterations of this project could look like. We also look into what technologies easily could have been changed in this current iteration to feature new possible experiments.

### 7.2.1 Scaling of Microservices

As this study does not account for the scaling of microservices, further experimenting with scaling microservices are needed. However, this should include multiple computers for both scaling the monolithic application, as well as the microservices, for a fair comparison. One should also be able to determine whether or not there exists a threshold for when microservices are more energy efficient and when the monolithic architecture is more efficient, and determine what this threshold is.

### 7.2.2 Remote Inter-Process Communication

In [10] S. Georgiou et al. the energy consumption of various inter-process communication (ICP from here) technologies like gRPC, RPC and REST are investigated. In this project, we only investigate the use of HTTP requests to communicate between the microservices. However, as a part of the future works, various ICP technologies could be implemented to investigate the impact that each have. S. Georgiou et al. compares the energy consumption of the ICP technologies across various CPU platforms and programming languages. To fit the study of S. Georgiou et al. to the study in this project, the focus would be on how the energy consumption of microservices would be impacted by interchanging the ICPs.[10]

### 7.2.3 External Client

In this project, all tests are conducted on a singular test computer that serves the clients (JMeter) as well as the services. In a future iteration of this project, it would be interesting to measure the individual energy consumption of the client as well as the server, on separated systems. This is also mentioned as a part of the threats to the validity in subsection 6.7.1.

### 7.2.4 Realism

One of the test design elements in this project, is that all our requests end up responding with a successful response. In a future iteration, it could be interesting to investigate the energy consumption of a more realistic system where requests have higher response time and some services might fail. It would be interesting to compare a monolithic and a microservice architecture when considering services potentially have to hold for other services due to failed calls or load-times. This also ties together with subsection 6.7.2 and subsection 6.7.1 that are mentioned in the threats to the validity of this project, and further researching this could lower the threats to the validity.

### 7.2.5 Architecture

Lastly, the obvious future iteration would be to extend this project to investigate other languages and other software architectures. The goal of this project was to investigate how microservices impact the energy consumption. However, other architectures, such as a serverless architecture, also needs to be investigated to fully develop a picture of what architecture is the best fit for a project that concerns with the energy consumption.

# Bibliography


[1] Omar Al-Debagy and Peter Martinek. "A Comparative Review of Microservices and Monolithic Architectures". In: *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. 2018, pp. 000149–000154. DOI: 10.1109/CINTI.2018.8928192.

[2] Amazon. *What is Docker?* Last retrieved 14-03-22. URL: https://aws.amazon.com/docker/.

[3] Vlad C. Coroama et al. "The Energy Intensity of the Internet: Home and Access Networks". In: *ICT Innovations for Sustainability*. Ed. by Lorenz M. Hilty and Bernard Aebischer. Cham: Springer International Publishing, 2015, pp. 137–155. ISBN: 978-3-319-09228-7.

[4] Marco Couto et al. "Towards a Green Ranking for Programming Languages". In: *Proceedings of the 21st Brazilian Symposium on Programming Languages*. SBLP 2017. Fortaleza, CE, Brazil: Association for Computing Machinery, 2017. ISBN: 9781450353892. DOI: 10.1145/3125374.3125382. URL: https://doi.org/10.1145/3125374.3125382.

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

[6] Cypress. *Why Cypress?* Last retrieved 02-06-22. URL: https://docs.cypress.io/guides/overview/why-cypress#Setting-up-tests.

[7] Docker. *Docker*. Last retrieved 02-06-22. URL: https://www.docker.com/.

[8] Docker. *Docker Stats*. Last retrieved 03-06-22. URL: https://docs.docker.com/engine/reference/commandline/stats/.

[9] Martin Fowler and James Lewis. *Microservices*. Last retrieved 25-05-22. 2014. URL: https://martinfowler.com/articles/microservices.html.

[10] Stefanos Georgiou and Diomidis Spinellis. "Energy-Delay Investigation of Remote Inter-Process Communication Technologies". In: *Journal of Systems and Software* 162 (Dec. 2019), p. 110506. DOI: 10.1016/j.jss.2019.110506.

[11] Daniel Guaman et al. "Performance evaluation in the migration process from a monolithic application to microservices". In: *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)*. 2018, pp. 1–8. DOI: 10.23919/CISTI.2018.8399148.

[12] Intel. *Intel Power Gadget*. Last retrieved 02-06-22. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html.

[13] Intel. *Running Average Power Limit Energy Reporting*. Last retrieved 02-06-22. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[14] Erik Jagroep et al. "The hunt for the guzzler: Architecture-based energy profiling using stubs". In: *Information and Software Technology* 95 (2018), pp. 165–176. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S0950584917303841.

[15] JMeter. *Apache JMeter*. Last retrieved 02-06-22. URL: https://jmeter.apache.org/.

[16] Alexander Kainz. *Microservices vs. Monoliths: An Operational Comparison*. Last retrieved 25-05-22. 2020. URL: https://thenewstack.io/microservices-vs-monoliths-an-operational-comparison/.

[17] Kubernetes. *What is Kubernetes?* Last retrieved 02-06-22. 2022. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[18] Craig Marcho. *Windows Performance Monitor Overview*. Last retrieved 02-06-22. URL: https://techcommunity.microsoft.com/t5/ask-the-performance-team/windows-performance-monitor-overview/ba-p/375481.

[19] Microsoft. *ASP.NET Application Life Cycle Overview for IIS 7.0*. Last retrieved 03-06-22. URL: https://docs.microsoft.com/en-us/previous-versions/bb470252(v=vs.140)?redirectedfrom=MSDN#life-cycle-stages.

[20] Middleware-Company.com. *The Petstore Revisited: J2EE vs .NET Application Server Performance Benchmark*. Last retrieved 20-05-22. URL: https://web.archive.org/web/20031203221937/http://www.middleware-company.com/j2eedotnetbench/.

[21] NGINX. *NGINX Docs*. Last retrieved 02-06-22. URL: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/.

[22] Daniél Garrido-Y Martinez Nielsen et al. *Energy Benchmarking With Doom - Utilizing video game source ports for macrobenchmarking*. Aalborg University, 2021.

[23] Wellington Oliveira et al. “Recommending Energy-Efficient Java Collections”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 160–170. DOI: 10.1109/MSR.2019.00033.

[24] Zakaria Ournani et al. “Evaluating the Impact of Java Virtual Machines on Energy Consumption”. In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM ’21. Bari, Italy: Association for Computing Machinery, 2021. ISBN: 9781450386654. DOI: 10.1145/3475716.3475774. URL: https://doi.org/10.1145/3475716.3475774.

[25] Cagri Sahin, Lori Pollock, and James Clause. “How Do Code Refactorings Affect Energy Usage?” In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’14. Torino, Italy: Association for Computing Machinery, 2014. ISBN: 9781450327749. DOI: 10.1145/2652524.2652538. URL: https://doi.org/10.1145/2652524.2652538.

[26] Vindeep Singh and Sateesh K Peddoju. “Container-based microservice architecture for cloud applications”. In: *2017 International Conference on Computing, Communication and Automation (ICCCA)*. 2017, pp. 847–852. DOI: 10.1109/CCAA.2017.8229914.

[27] Freddy Tapia et al. “From Monolithic Systems to Microservices: A Comparative Study of Performance”. In: *Applied Sciences* 10.17 (2020). ISSN: 2076-3417. DOI: 10.3390/app10175797. URL: https://www.mdpi.com/2076-3417/10/17/5797.

[28] Mario Villamizar et al. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”. In: *2015 10th Computing Colombian Conference (10CCC)*. 2015, pp. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476.

# Appendix A

# JMeter configuration

The Ultimate Thread Group setup is specified in Table A.1 and A.2.

| **Start threads count** | 25 | 50 | 75 | 100 | 150 | 200 | 250 | 300 |
|---|---|---|---|---|---|---|---|---|
| **Initial delay, sec** | 0 | 450 | 900 | 1350 | 1800 | 2250 | 2700 | 3150 |
| **Startup time, sec** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **Hold load for, sec** | 7199 | 6749 | 6299 | 5849 | 5399 | 4949 | 4499 | 4049 |
| **Shutdown time** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Table A.1:** Clients Ultimate thread group configuration.

| **Start threads count** | 350 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 |
|---|---|---|---|---|---|---|---|---|
| **Initial delay, sec** | 3600 | 4050 | 4500 | 4950 | 5400 | 5850 | 6300 | 6750 |
| **Startup time, sec** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **Hold load for, sec** | 3599 | 3149 | 2699 | 2249 | 1799 | 1349 | 899 | 449 |
| **Shutdown time** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Table A.2:** Clients Ultimate thread group configuration continued.

# Appendix B

# Result comparison

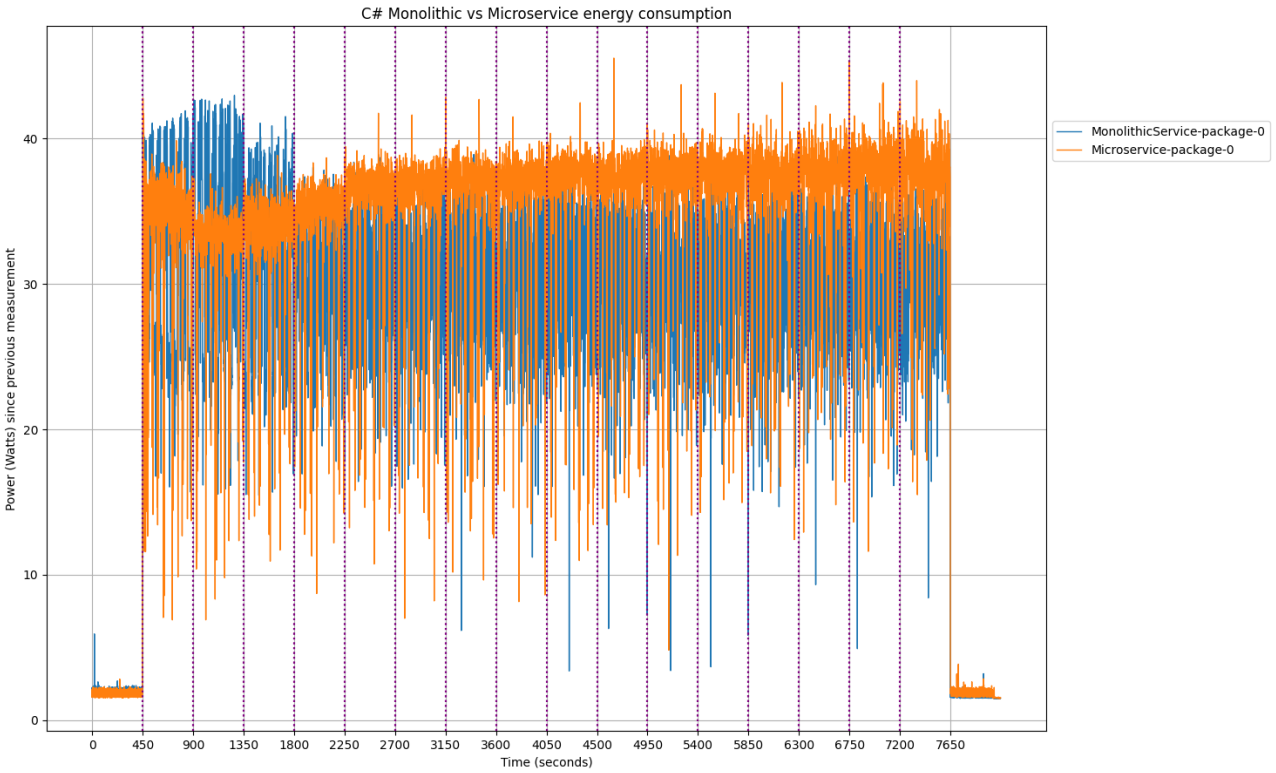



**Figure B.1:** Overlap of energy consumption results from C# microservice and monolithic service implementation tests.

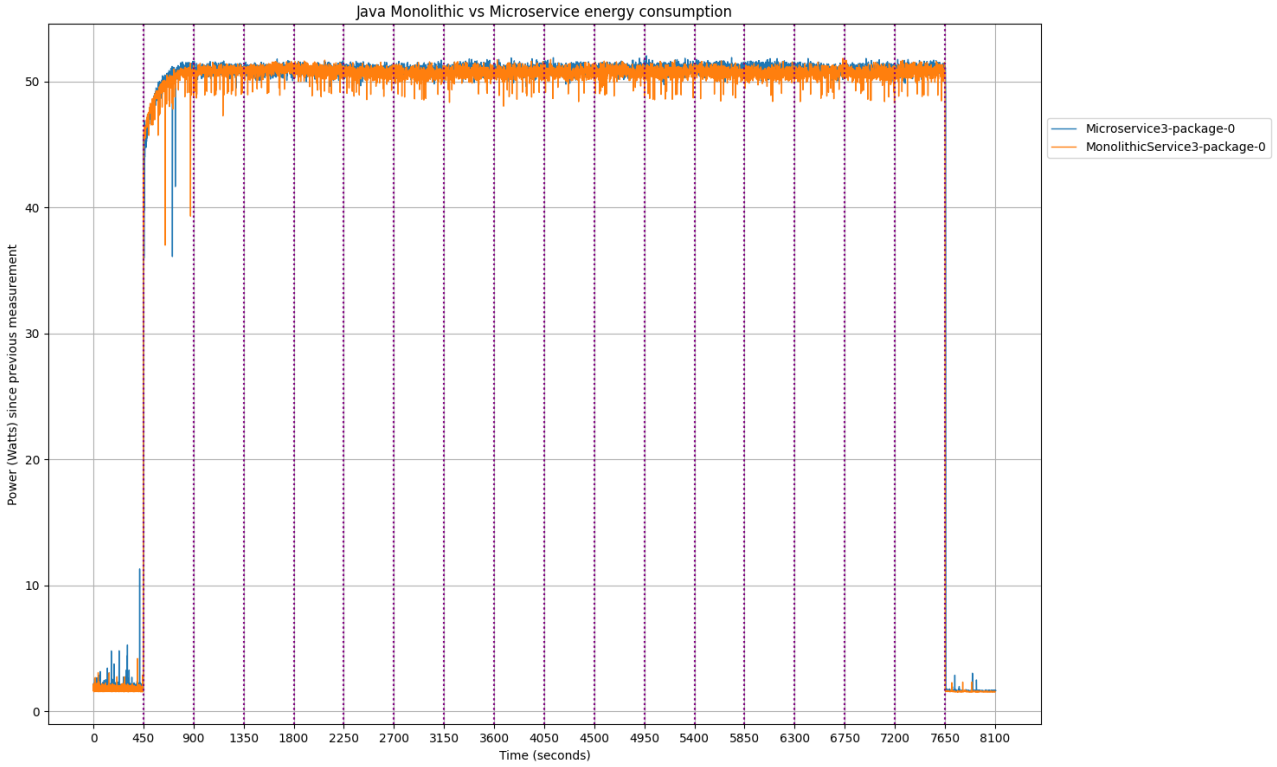



**Figure B.2:** Overlap of energy consumption results from Java microservice and monolithic service implementation tests.