Sustainability: Practical energy efficiency improvements in software development

Software Engineers:

Eduard Conesa Guerrero eduard.conesa-guerrero@capgemini.com

Airam Hernández Rocha airam.hernandez-rocha-external@capgemini.com

Fabián Scherle Carboneres fabian.scherle-carboneres@capgemini.com

Maximilian Hammer maximilian.hammer@capgemini.com

Project Coordinator:

Ignacio Gallego Sagastume ignacio.gallego-sagastume@capgemini.com

Main Architect

Santos Jiménez Linares santos.jimenez-linares@capgemini.com

Executive Sponsors: Francisco Bermúdez Atalaya, Thilo Hermann

Product Designer & Owner: Iwan van der Kleijn

Table of Contents

Abstract

Energy consumption is strongly associated with computing performance. Improvements are not only achieved by optimizing code, but also by other factors such as the choice of programming language. For many enterprises, reducing CO2 emissions is seriously important and a source of concern.

This report studies three different approaches for reducing energy consumption:

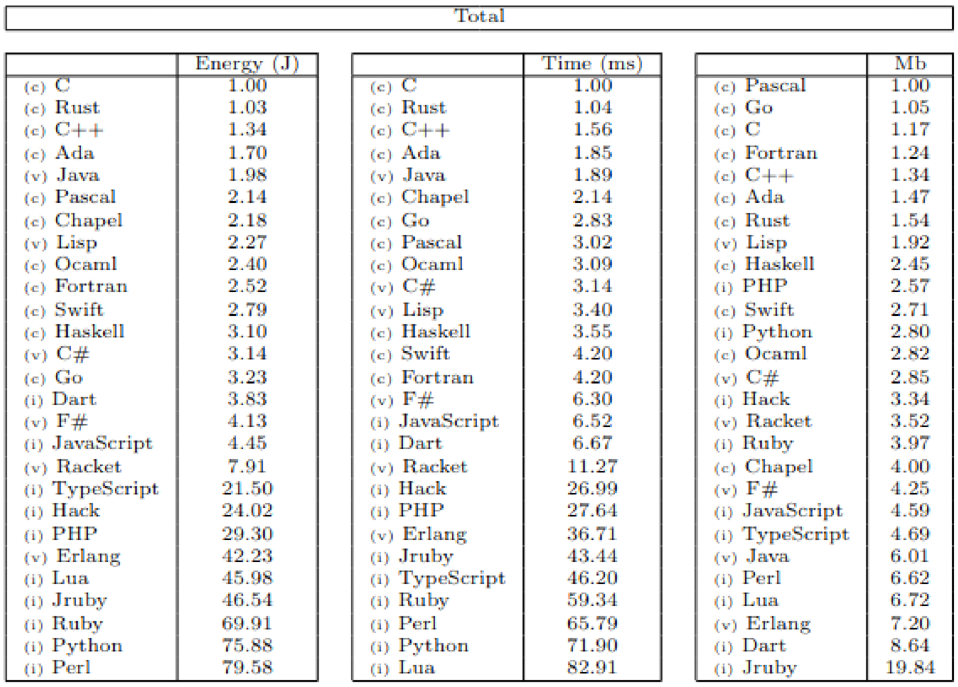
1. Adopting highly optimized programming languages
2. Changing to architecture topologies that are inherently more efficient
3. Moving to Cloud Hyperscalers

Two types of benchmarks are used to test an example application, developed on 5 different languages. The first one checks the energy consumption when the program is subjected to a constant load and the second one tests the number of requests that it can withstand.

Then, the application with the most performant language is adapted to an “Edge Computing” architecture. Different approaches show how decisions during development affect performance and thus, energy consumption.

State of the art

The difference in energy consumption between languages is quite significant, but it was not measured accurately until a few years ago. There are some interesting results in the study “Ranking Programming Languages by Energy Efficiency” [1]. The average results from the same study obtained in the 2017 version [2] are shown in the next table:



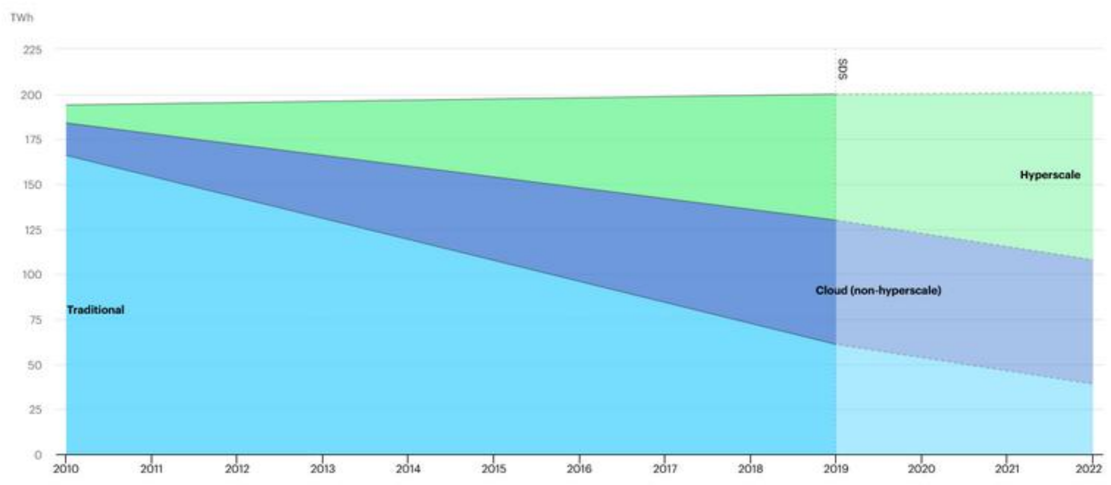
Total					
	Energy (J)		Time (ms)		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Sourced: “Energy efficiency across programming languages: how do energy, time, and memory relate?” Normalized global results for Energy, Time, and Memory. Pereira et al. 2017

The left column is the energy consumption in Joules, the center represents the time in milliseconds and the right one, the RAM in Mb.

Every column takes the best performing language as the base for the comparison and indicates how many times more every other language consumes in each category given an average task. For example, in the time column, Pascal takes, in average, 3 times more time than C for the same task.

Additionally, it is incredibly counter-intuitive given the tremendous growth of big data, machine learning, edge devices, and even the growth of scale of software in general experienced by our industry over the last 10 years, that global software energy consumption is constant at first glance, as can be seen in the next graph [3].



Source: IEA (2021), Global data centre energy demand by data centre type, 2010-2022, https://www.iea.org/data-and-statistics/charts/global-data-centre-energy-demand-by-data-centre-type-2010-2022. All rights reserved.

The energy consumed by traditional data centers (blue section) is being reduced and overtaken by cloud providers, hyperscalers (green) and non-hyperscalers (purple). It can be interpreted as if cloud providers were simply replacing the traditional option without any improvement, but we must consider that software scale, usage and quantity has increased immensely. This means that if cloud providers had the same efficiency as traditional data centers, energy consumption would have increased exponentially.

Objectives

This project studies a set of proposals directed at drastically reducing energy consumption of software by:

1. Using highly optimized programming languages and runtime environments.
2. Choosing architecture topologies which are inherently more energy efficient.
3. Moving to Cloud Hyperscalers which are intrinsically more energy efficient as compared with on-premises or other Cloud Data Centers.

Scope

To do the experiments, a base application was developed in five different programming languages with a standardized response structure and HTTP status for every endpoint.

The application is called “Jump the Queue” ("JTQ" from now on, see [6]) and it is a REST API application used for the onboarding process at Capgemini. It controls an event’s queue by giving each visitor a code, representing their number in line just like in a supermarket or similar. The actions performed by the user are:

- Register
- Login
- Join a queue
- Show the queue
- Leave the queue.

All details of JTQ API are covered in the wiki of the Java version. The version used in the study has some changes for performance and functionality purposes, such as the generation of ticket numbers.

JTQ API

The API has the next URL base: /jumpthequeue/services/rest

The endpoints that all backends implement are:

Queue

Req	Res Status	URL	Description
GET	200/404	/queuemanagement/v1/queue/ID/	Get queue by ID
POST	200	/queuemanagement/v1/queue/	Create a queue
POST	200	/queuemanagement/v1/queue/search/	Search queues with a criteria
DELETE	200/404	/queuemanagement/v1/queue/ID/	Delete a queue

Visitor

Req	Res Status	URL	Description
GET	200/404	/visitormanagement/v1/visitor/ID/	Get visitor by ID
POST	200	/visitormanagement/v1/visitor/	Create a visitor
POST	200	/visitormanagement/v1/visitor/ID/search/	Search visitors with a criteria
DELETE	200/404	/visitormanagement/v1/visitor/ID/	Delete a visitor

Access Code

Req	Res Status	URL	Description
GET	200/404	/accesscodemanagement/v1/accesscode/ID/	Get access code by ID
POST	200	/accesscodemanagement/v1/accesscode/	Join a visitor to a queue
POST	200	/accesscodemanagement/v1/accesscode/search/	Search access code with a criteria
DELETE	200/404	/accesscodemanagement/v1/accesscode/ID/	Remove a visitor from a queue
GET	200	/accesscodemanagement/v1/accesscode/cto/ID/	Get the compound version of an access code
POST		/accesscodemanagement/v1/accesscode/cto/search/	Search access code with a criteria in the compound version

Approaches

Different programming languages and frameworks

Two main factors affect the energy consumption, efficiency and performance of a programming language. Whether it is compiled, semi-compiled (virtualized) or interpreted and the programming paradigm it is based on.

Considering these aspects in conjunction with the abilities and experience of the team members, the chosen programming languages and frameworks were:

- Java with Spring Boot
- Rust with Actix 4
- TypeScript with NestJS
- Python with Django
- C# with ASP .NET
- Web Assembly compiled from Rust using the Atmo Framework

Distributed architecture topologies

In most modern applications, loading times are more than ever related to network traffic instead of real processing time. The increase in speed of internet connections has helped with this problem, but even light speed has a limit and for use cases that require sending and receiving large amounts of data (video streaming is a clear example). That is, for example, it is not efficient to require a user to always access a server in another continent.

However, data transmission is not the only concern. All computing devices have gotten much more powerful in the last decade, but they are mostly underutilized because the main processing is almost always done in the server side.

Edge computing is an architecture paradigm that aims to solve both these problems by moving all the data and processing possible as close to the end user as possible, even to the user's own device which in most cases has a highly efficient CPU. Doing this will allow for faster applications, will address the problem of the massive unused compute capacity and greatly reduce network traffic and data center's load and dependency on them.

Because of the problems this architecture is trying to solve, it shines the most when applied to two types of applications:

1. Mostly read only applications, where write operations are either infrequent or where consistency is not a priority, such as Netflix or YouTube.
2. Data processing applications, where after making the necessary operations, the information is stored and not consumed by other clients. A clear example is IoT systems like an application that manages your home devices such as an intelligent thermostat or a robot vacuum, that need to process their environment to give you a better experience, but that data is not queried by other users.

To test these assumptions, we have adapted the test application to, instead of using a monolithic server, use an "edge server" meant to be deployed in locations close to the user, acting as a big cache storage, and a "central server", that will be in a normal data center, not necessarily close.

A standardized benchmark will be executed in two situations:

- Only the central server deployed
- Central and edge servers deployed

This will show if having a server near the user improves performance and CO2 emissions.

Energy consumption in this case is not comparable with the results in obtained in the local machines because of the differences in environment and accessible data. The tools used to measure consumption cannot be applied in virtualized environments, so results obtained in this section can only be compared between them.

Differences with the monolithic implementation

As previously mentioned, Edge Computing architecture improves read operations and data processing performance. JTQ is not, however, the type of application that could benefit the most from it, because consistency is a priority. However, even if write operations are not improved, the GET endpoints take advantage of the local cache in the Edge Server, highly reducing network traffic and loading times.

This is how the edge server operates:

1. When obtaining a single element, if it is present in the local database, send it to the client. If not, make a request to the central server and cache it before sending the response.
2. All write operations will cache the data in the local database and send it to the central server

3. Search operations need to look in all the data in the application. The edge server will make a request to the central server and cache all the response data.
4. To keep eventual consistency, if an element is stored in the local database without being updated for more than X minutes it will be deleted.

As it can be seen in the changes, other types of applications will benefit much more from this architecture, but our testing will reflect both the major strengths and weaknesses of its implementation.

Moving to Cloud Hyperscalers

Measuring power consumption in AWS

There are multiple services offered by AWS with different access to the underlying hardware. In a virtual machine, tools to measure the power consumption of the processor and memory like powertop or Intel's RAPL are widely available. Due to unknown reasons, those tools do not report the power usage when using an AWS EC2 instance. Using other PaaS or SaaS services, like AWS Elastic Container Service (ECS), the measurement or estimation of power consumption shows the same difficulty. Measuring power consumption of specific containers or clusters in ECS is unfortunately impossible for users directly.

Due to Amazon's goal to achieve carbon neutrality by 2040, customers can make use of a sustainability dashboard. Unfortunately, this dashboard does not show the direct power usage, it only provides the amount of greenhouse gas emissions as the equivalent amount of carbon dioxide. The data is provided on a monthly basis and has an up to three months lead time.

Measuring power consumption is, as described above, impossible for AWS customers. The only option left is to estimate the power consumption. There are multiple ways to estimate the power consumption of a cloud installation. The most suitable approach for the use case of comparing languages is to find the CPUs rated power consumption as a base line. The so called rated TDP (thermal design power) is then divided into the number of possible vCPUs. Due to AWS' way of splitting up CPUs, this number is equal to the number of logical threads the processor provides. In the next step, it is possible to see the number of vCPUs used. Using AWS' Cloud Watch this data can be gathered, alternatively a background process can record the CPU utilisation, which can be converted to the number of vCPUs used. Using the formula:

$$\frac{rated\ tdp}{number\ of\ possible\ vCPUs} \times \frac{allocated\ vCPUs}{number\ of\ possible\ vCPUs} \times usage\ vCPUs,$$

allows for an estimate in power consumption for a ECS cluster or EC2 instance.

Unfortunately, there is no way of measuring or estimating the power consumption of memory. There are vast differences in the memory management of the tested languages. Measuring the impact of these approaches on the power consumption of memory would be an interesting comparison. However, the power consumption of memory is, according to the authors of [1], negligible in comparison to the power consumption of the overall system and therefore not important to measure.

Methodology

To test the performance and consumption of the different programming languages following the previously mentioned approaches, we have implemented two benchmark programs using Rust with the Goose framework.

Both benchmarks are configured to perform a sequence of GET, POST and DELETE HTTP requests to the respective endpoints. The first benchmark seeks to only test energy consumption by sending 14 requests per second for 5 minutes. The second benchmark tests the consumption related to the peak performance by sending as many requests as possible. When each test ends, the results are generated in HTML format, which are processed to remove information that is not useful or is redundant, such as the status code for individual requests.

Measuring tools

Local tests

The Energy consumption is measured using a tool called "Intel Power Gadget" [4], which can be launched as a console application or by script. As its name suggests, it only works in computers with an Intel CPU. In this case, it starts running at the same time as the benchmarks. When the testing process ends, a file in CSV format is generated containing the results.

Cloud tests

In AWS, given its previously mentioned limitations, estimating the power consumption was more complex.

To compare the Edge Computing architecture to the monolithic architecture, firstly we developed a mini-API that will only receive one request: start PowerTop with the running time and results file name passed as parameters. Secondly, unlike the local tests (where we used specific benchmarks written in Rust and Goose), for the distributed architecture tests we developed a generic benchmark that can be customized using a YAML configuration file, where 9 out of 10 requests are GET requests.

With the test script, we launch PowerTop in both EC2 instances and start the benchmark. Then, with the CPU usage results, we approximate the consumption using the formula presented above.

To compare the local consumption against the cloud consumption, instead of PowerTop we opted to use Cloud Watch to get the mean consumption while the benchmark was running. This is because in the EC2 instance where these tests were executed, idle consumption was negligible, not accounting for even 1% of CPU usage.

Test flow

Different programming languages

The testing process is carried out with a machine that has a Windows 10 operating system with a quadcore Intel i5 1135G7 at 2.4GHz with Hyper-Threading, along with 8 GB of RAM. The Docker runtime is handled by Rancher Desktop [5]. The chosen tool for energy consumption measurements is, as previously mentioned, Intel Power Gadget.

Having the components implemented, the tests are executed, and the data is generated through bash and PowerShell scripts, considering the idle consumption of the system and of each process once the benchmarks are launched. The final results are detailed in the "Results" section.

Our test flow consists of the following steps:

1. Measure the energy consumption of the idle machine for 5 minutes
2. Wait for 60 seconds to get a stable state
3. For each programming language
    a. Start a fresh instance of the database
    b. Start the application (locally or in a Docker container)
    c. Wait for 2 minutes for the system to stabilize
    d. Execute the constant load benchmark
    e. Execute the maximum load benchmark
    f. Close the application
    g. Close the database
    h. If there are any remaining languages, wait for 5 minutes to get a stable system again
4. Process the generated data

Distributed architecture topologies

Our EC2 instances for the central and edge servers were running Ubuntu 22.04 with an Intel Xeon E5-2676 at 2.4GHz, that has a TDP of 120W and from which we only used a single core. The equipped RAM was 1GB. They were instantiated Virgina, USA and Nothern Ireland respectively.

The test flow for the edge architecture consisted of these steps:

1. Make a request to the PowerTop management server with a 5 second margin over the benchmark duration.
2. Start the benchmark, making 14 requests per second to the edge server.
3. Once the benchmark finishes and PowerTop stops measuring, calculate the consumption of both, the central and edge server and put it together.

To approximate the consumption of only having a central server in Virginia to compare it against the edge architecture, the same steps are followed except for the final addition.

Moving to Cloud Hyperscalers

To compare against the local consumption, we used a Nothern Ireland EC2 instance with an Intel Xeon Platinum 8175M with 8 virtual cores assigned, along with 32GB of memory.

Out test flow for this comparison is the following:

1. Run the benchmark 1 or 2.
2. Get the mean consumption in the time that the benchmark was running using CloudWatch.
3. Use the formula presented in the “Measuring power consumption in AWS” section to calculate the energy consumed.

Results

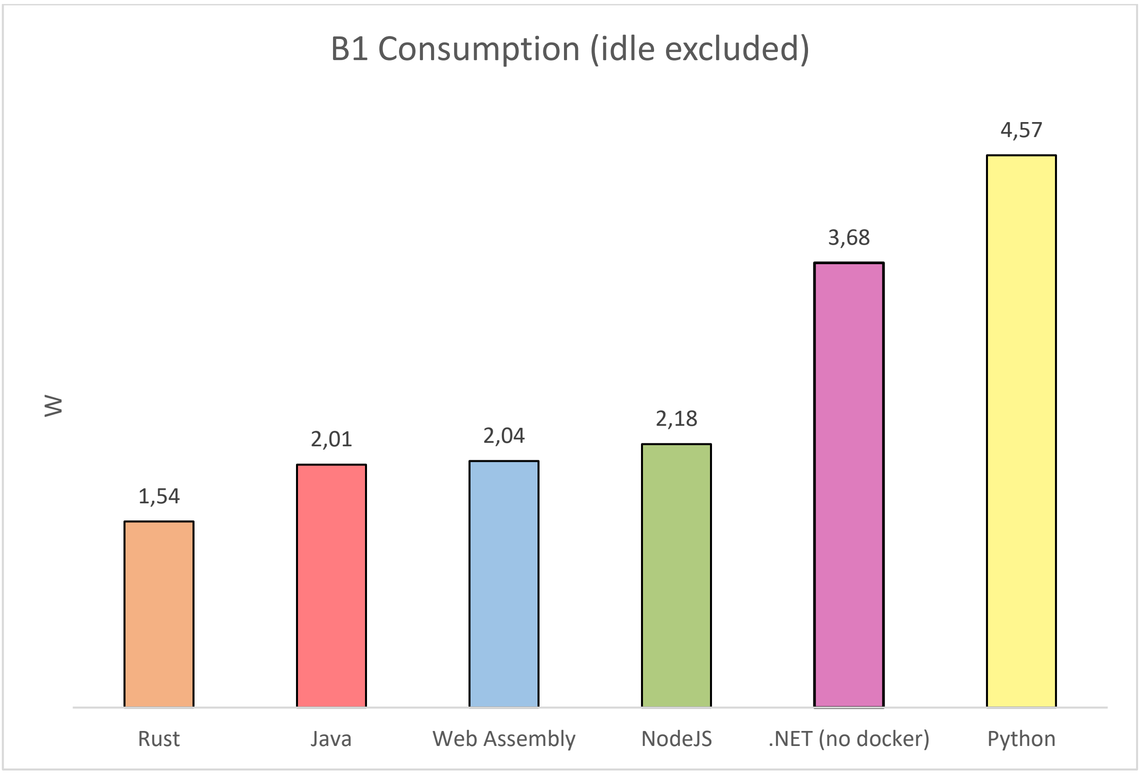
Different programming languages and frameworks

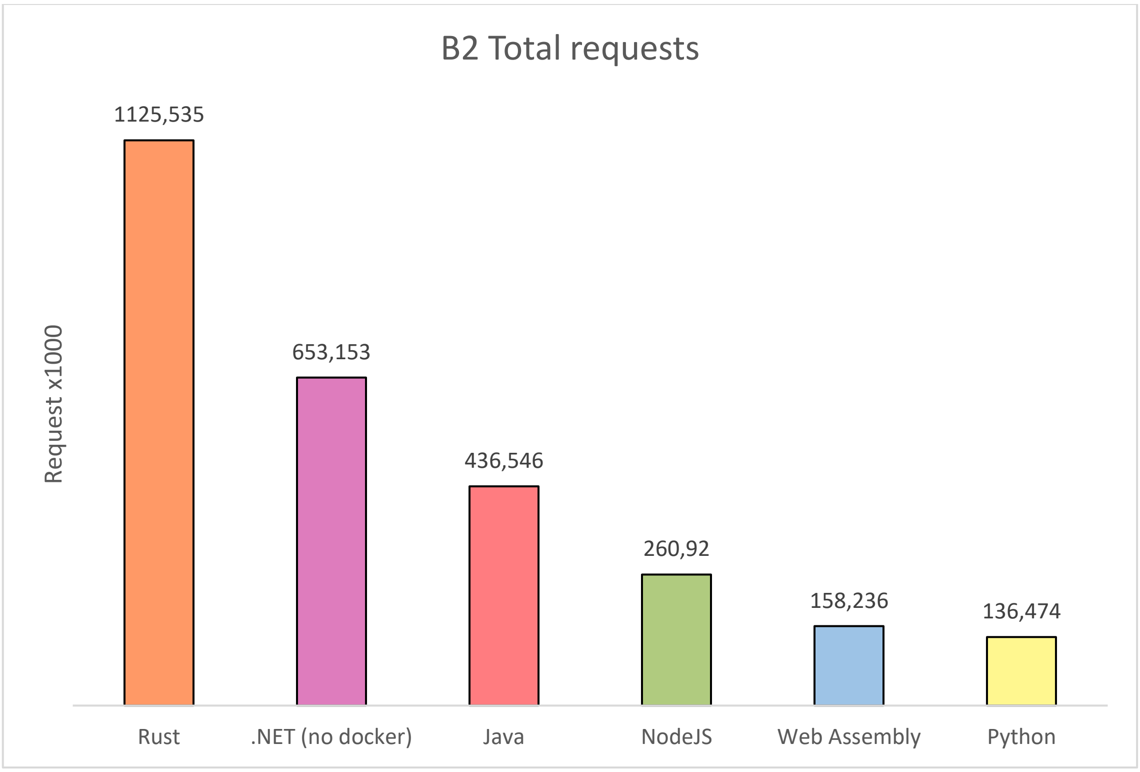
The results were mostly expected considering previous studies and the category of each language (compiled, semi-compiled and interpreted). Rust as the only purely compiled language was the leader in every test.

In terms of efficiency under constant load, the most unexpected result was the abnormally high consumption of .NET 5 when compared with Java or even NodeJS, which can be attributed to a higher base consumption of its runtime environment.

In the more demanding test, Rust was the leader almost doubling the number of requests of the next language, surpassing the million in 5 minutes. The only changes from the previous ranking are .NET 5 surpassing Java with 50% more attended requests and Web Assembly not being able to keep up in performance with the compiled and semi-compiled languages. This test is also extremely important from an energy consumption point of view because in situations of high traffic, languages like Java or Python would require additional instances of the application, consuming more and impacting costs, while Rust could be able to handle the peak without any need for extra scale.

In the specific case of Web Assembly, it needed extra instances like Python or NodeJS managed with NGINX to handle the load and make use of all the available threads, because of its single threaded nature. There is a very important difference however, in consumption when comparing it with pure Rust, even if for the programmer, the development experience is practically the same. With this example it is absolutely clear that more than the language itself, the compilation target is what matters the most in performance and energy consumption.



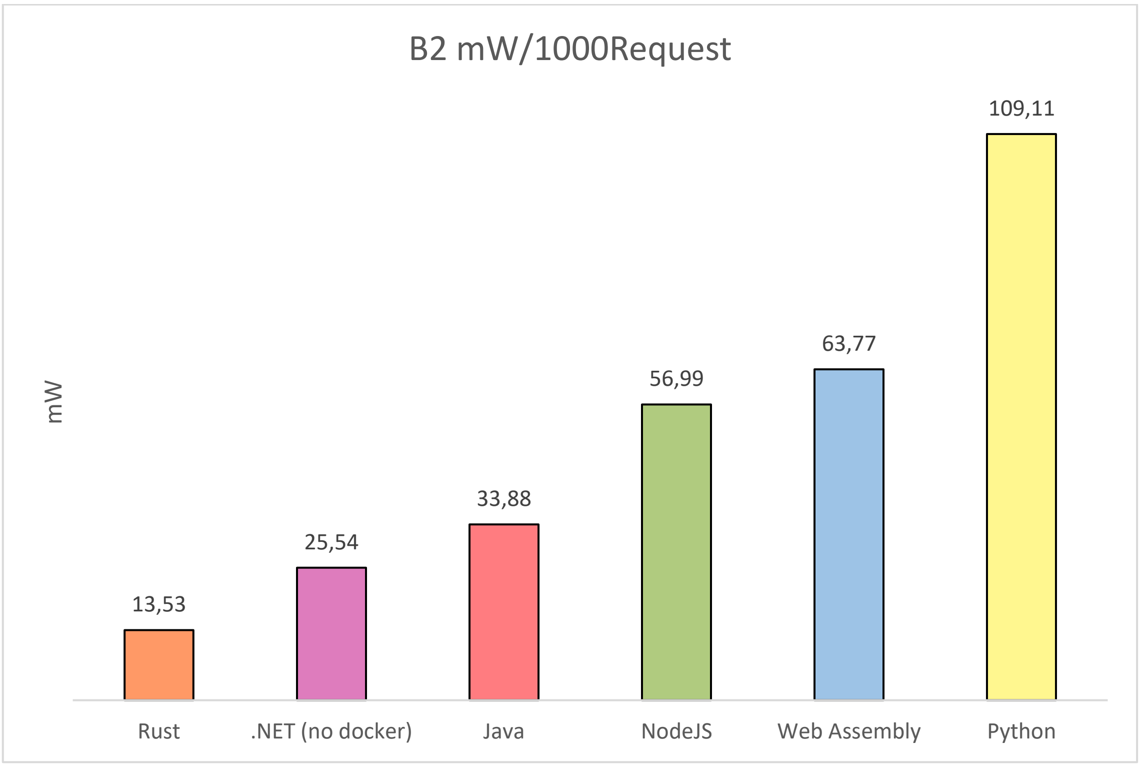


When putting the languages at their performance limit with this hardware, we can see that of course, the overall consumption is the mostly the same (except for a slightly lower value in the case of Web Assembly, which can indicate that is not making proper use of the full potential of the machine), because the processor is at its full potential. The main difference comes when relating the overall consumption with the number of requests made.

	W(idle excluded)
Rust	15,23
.NET (no docker)	16,68
Java	14,79
NodeJS	14,87
Web Assembly	10,09
Python	14,89

Table 1. Total consumption in benchmark 2

When relating the performance with the total number of requests it is clear that Rust is the most efficient language. This means that, for example, in a Cloud environment where the instances scale with load, if our program is written in Rust, it will always need less instances to handle the same load and will consume less energy, reducing emissions and saving us money.


B2 mW/1000Request
mW
13,53
25,54
33,88
56,99
63,77
109,11
Rust
.NET (no docker)
Java
NodeJS
Web Assembly
Python

Distributed architecture topologies

The next tables show the conversion from CPU usage to power consumption assuming a TDP of 5W, which we get from dividing the total TDP of the Intel(R) Xeon(R) CPU E5-2676 v3 2.40GHz (used by the EC2 instance) between its number of virtual cores (24 threads and 120W of TDP).

We highly suspect that AWS is limiting the number of requests per second we can send from a single host, which could be hindering the results from the Benchmark 2 not giving us a realistic performance result.

Edge_B1	usage (us/s)	% cpu	Power (mW)
backend_edge	462,6	0,04626	2,313
DB_edge	159,6	0,01596	0,798
backend_central	153,5	0,01535	0,7675
DB_central	179	0,0179	0,895

Edge_B2	usage (us/s)	% cpu	Power (mW)
backend_edge	688,2	0,06882	3,441
DB_edge	165,8	0,01658	0,829
backend_central	311,4	0,03114	1,557
DB_central	202,7	0,02027	1,0135

The same results are calculated for the normal Rust JTQ API.

Monolitic_B1	usage (us/s)	% cpu	Power (mW)
backend	416,8	0,04168	2,084
DB	393,2	0,03932	1,966

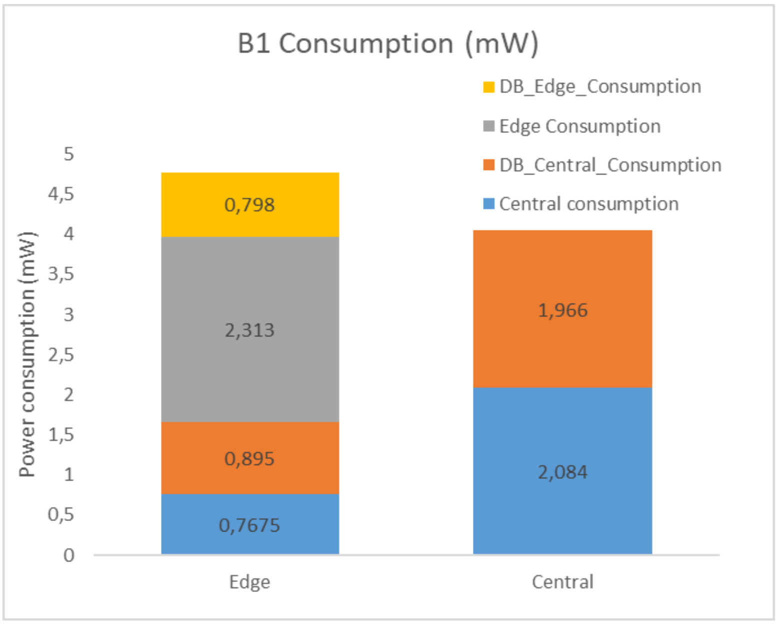
Monolitic_B2	usage (us/s)	% cpu	Power (mW)
backend	388,6	0,03886	1,943
DB	415,5	0,04155	2,0775

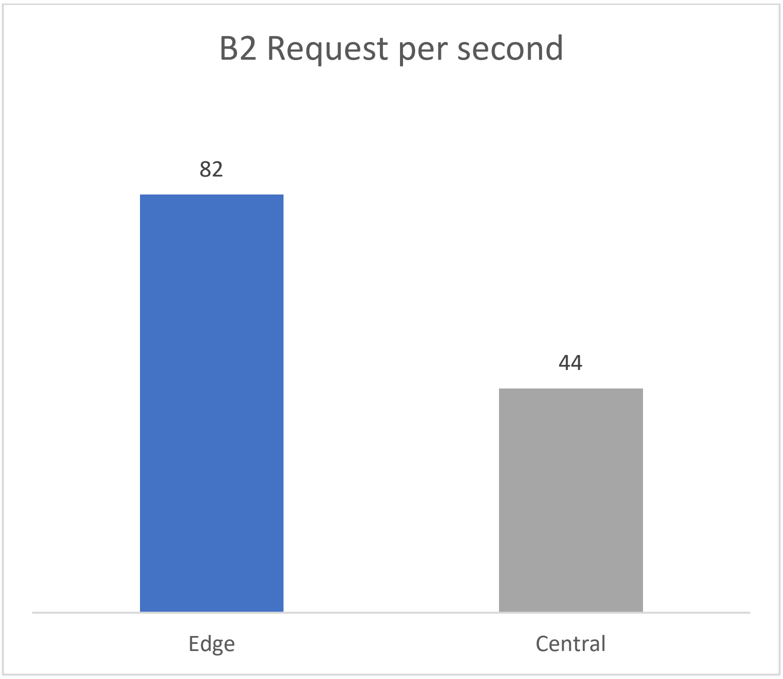
To sum up, in the next table we can see the power consumption and performance for both benchmarks in the two architectures, Edge Computing and a centralized server.

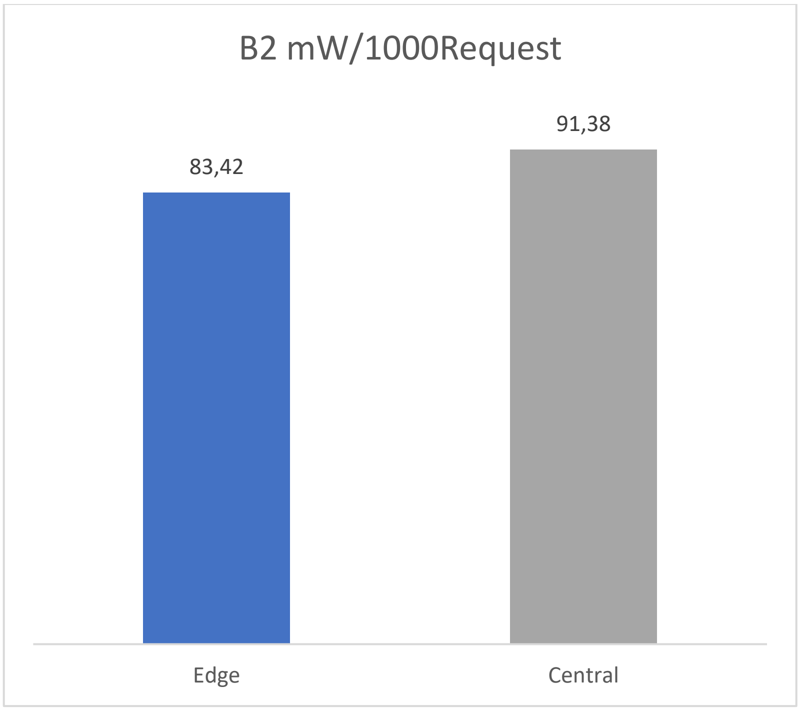
		Edge		Central		
Edge	Req/s	Backend (mW)	DB (mW)	Backend (mW)	DB (mW)	Total (mW)
B1	14	2,313	0,798	0,7675	0,895	4,7735
B2	82	3,441	0,829	1,557	1,0135	6,8405

Mono	Req/s	Backend (mW)	DB (mW)	Total (mW)
B1	14	2,084	1,966	4,05
B2	44	1,943	2,0775	4,0205

Only looking at the benchmark 1 overall consumption graph, it would seem like the edge version consumes more, but when increasing the load with the benchmark 2 we can see that, as expected, the Edge Computing architecture is better than the centralized server, both in performance as well as in energy efficiency: It delivers double the performance with a smaller consumption per request ratio.



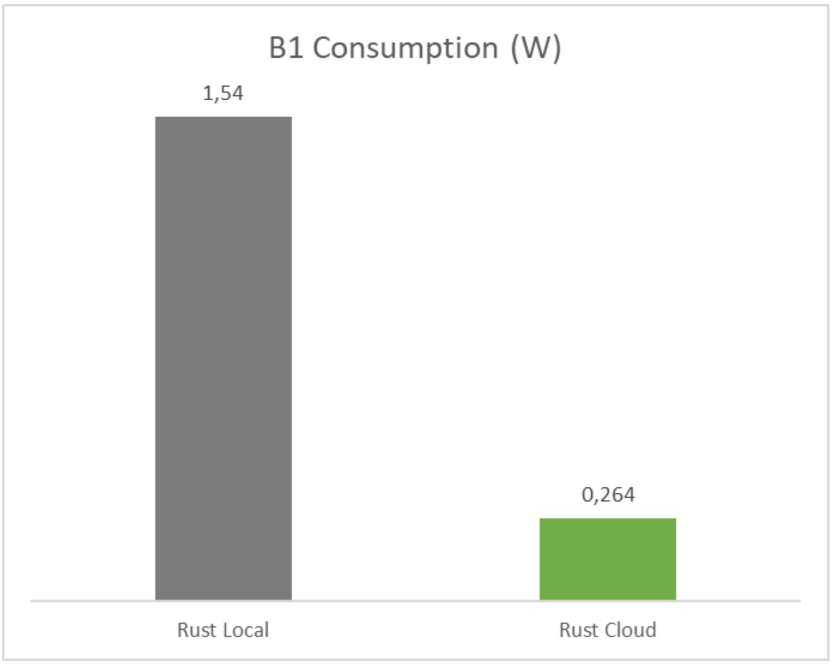




Moving to Cloud Hyperscalers

When running the benchmark 1, consumption is reduced almost 6 times compared to the results obtained in our local machines, and that is even taking into account idle consumption.

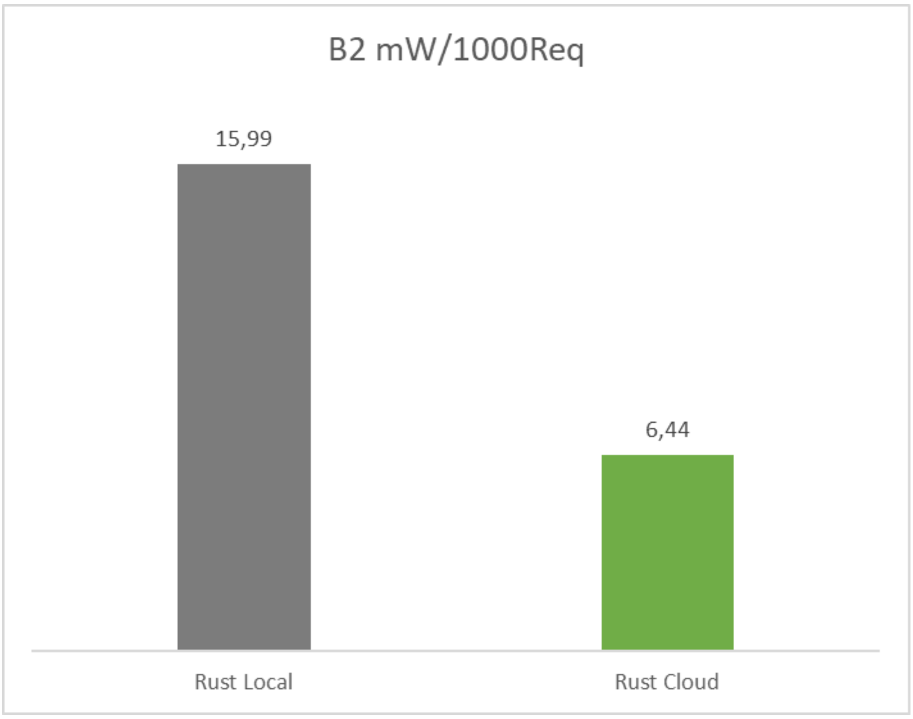
B1	% cpu	Power (mW)
backend	0,88	264



This comparison is to be taken with a grain of salt, given the vastly different hardware and measuring methods. The cloud results, contrary to the local ones, are estimated as we explained earlier because of the high difficulty of measuring consumption in an AWS instance and the reduced toolset these platforms offer regarding this kind of data. The local numbers are taken from the tests performed in the "Different programming languages" approach, which were executed in the laptop specified under "Methodology - Test Flow – Different Programming Languages".

	Rust Local	Rust Cloud
B2 Consumption (W)	18	17,39
B2 Req	1125535	2700000
B2 mW/1000Req	15,99	6,44
B2 Req/s	3751,78	9000,00

With the benchmark 2, we can clearly see that the cloud version can deliver more than double the performance practically the same consumption, resulting in less than half the consumption per request.



Future Work

For future works we would like to measure the power consumption in a data center where we have access to the hardware and can measure the energy consumed using the same tools as in our local machines, both to make a much more reliable comparison and to test the accuracy of our estimations.

In addition, try to measure impact in energy consumption of the elements surrounding the machine itself, such as cooling or networking, that should constitute a considerable part of the overall data center’s consumption.

Conclusions

In this study we have seen that from the three tested approaches, the one that reduces consumption the most is the election of programming language. Modern compiled languages will, in almost every case, surpass every semi-compiled and interpreted language in terms of performance and energy consumption. Web Assembly is still very infant as a compile target for backend development and lacks the tools and performance to justify using it in its current state. Rust, with its increase in popularity and the energy efficiency and memory safety that it provides, will be the most future proof bet for almost every new project.

The chosen architecture affects mainly the performance, but moving more of this processing to the client (which is the final objective) and taking into account the consumption derived from the traffic, it will be much more efficient in the long run. Although, you should take into account if your application really fits in one of the best-case scenarios that we explained for this architecture.

Regarding the cloud hyperscalers, we have seen that the consumption for the same load is incredibly low in comparison, and that is using a fixed EC2 instance. If you also consider the elastic load services and how easy is to move your applications to data centers that are carbon neutral, deploying in the cloud instead of in your own data centers is a no-brainer.

But even with this order, with existing applications is often not possible to change the programming language or even the topology, so which one to take is a per-project discussion. Our recommendations are as follows:

1. For new projects, consider using highly optimized programming languages, it will save energy and therefore money.
2. If your project fits in the best-case scenarios for Edge Computing, consider implementing it or migrate your application to adapt to it.
3. If you have to choose between deploying in a self-managed data center or in the cloud, the second one will almost always be a better option.
4. For existing and flexible projects where changing languages is not a possibility, and fit the best-case scenarios for Edge Computing, consider that migrating your application could increase its performance, better the user experience, save energy and money.
5. For projects that cannot change their programming language nor topology, deploying in the Cloud in the best way to reduce their energy consumption.

Bibliography

[1]

Ranking programming languages by energy efficiency

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, João Saraiva (2021),

Science of Computer Programming, Volume 205, 2021, 102609, ISSN 0167-6423,

https://doi.org/10.1016/j.scico.2021.102609.

[2]

Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, João Saraiva (2017),

Association for Computing Machinery, Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, ISBN 9781450355254,

https://doi.org/10.1145/3136014.3136031

[3]

Global data centre energy demand by data centre type,

IEA, Paris (2010-2022),

https://www.iea.org/data-and-statistics/charts/global-data-centre-energy-demand-by-data-centre-type-2010-2022

[4]

Intel Power Gadget,

Joe Olivas, Timo Kleimola, Mark Price, Timothy McKay,

Intel,

https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html

[5]

Rancher Desktop,

SUSE Team,

https://rancherdesktop.io/

[6]

Jump The Queue API

ADCenter Valencia, Capgemini (2022)

https://github.com/devonfw/jump-the-queue/wiki