Writing greener Java applications

How Quarkus can help reduce carbon footprint and improve sustainability

Table of contents

facebook.com/redhatinc
@RedHat
linkedin.com/company/red-hat

Executive summary

- There is an urgent need to reduce the impact of the software industry on the environment. This is a solvable problem, and it is one within the control of individuals designing and writing applications.
- Improving efficiency is a no-regrets solution to reduce carbon emissions while lessening cloud costs and improving user experience.
- Quarkus significantly reduces the resource usage of Java(TM) applications, while also improving developer productivity. Expect energy consumption to be reduced by a factor of two or three compared to legacy cloud-native frameworks.

Sustainability and the IT industry

Improving sustainability has a range of business and social benefits. It can reduce costs, help attract businesses attract customers, and act as a recruitment differentiator. In a world increasingly being shaped by harmful climate change, it is also its own reward.

Sustainability is particularly relevant in the IT industry, since this industry has a disproportionate environmental effect. As well as consuming energy, much of it fossil-fuel derived, the ICT industry also consumes valuable abiotic resources such as water and minerals. Finally, IT equipment is often associated with pollution at the beginning of its life cycle, during manufacturing, and e-waste at the end of its life cycle.

Carbon emissions from software

Climate change is a pressing challenge that requires individual, organizational, and government action. The 2022 IPCC Sixth Assessment Report estimates that to restrict the earth's warming to 1.5° above pre-industrial levels, we must reduce greenhouse gas emissions by 45% by 2030.

ICT has a part to play in this reduction, because the energy consumption of the world's computers is significant. One study estimated the digital world consumed 5.5% of the world's electricity, and is responsible for 3.8% of its greenhouse gas emissions. This is significantly more than aviation, which accounts for 1.9% of the world's greenhouse gas emissions.

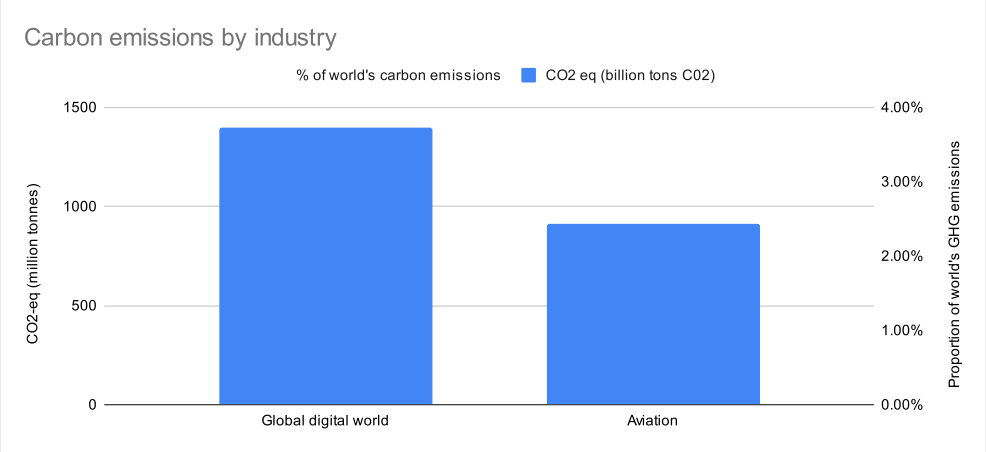


Figure 1. Industrial emissions comparison [1][2]

Here, "digital world" includes end-user devices, networks, and datacenters. What if we consider just datacenters?

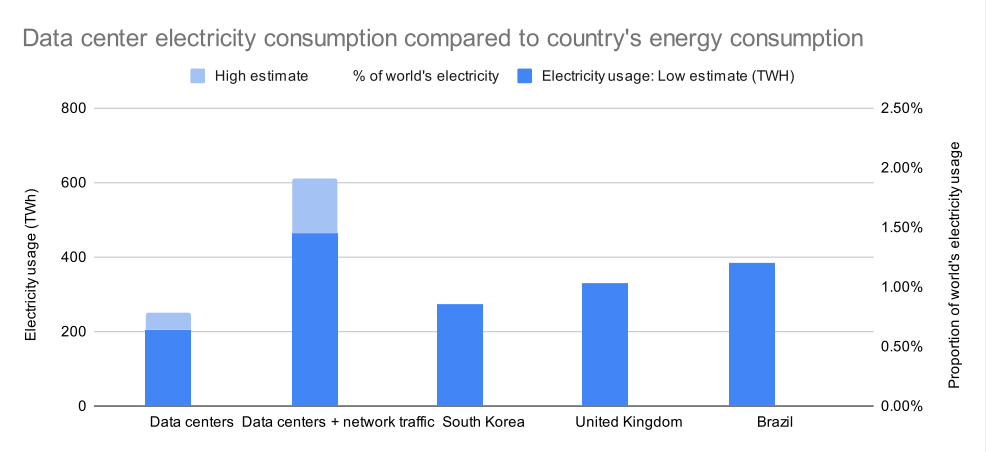


Figure 2. Electricity consumption comparison.[3][4] Lighter area represents high and low estimates, where available.

1 *"Environmental footprint of the digital world."* GreenIT.fr, November 2019.

2 *Ritchie, Hannah. "Sector by sector: where do global greenhouse gas emissions come from?" Our World in Data, Sept. 18, 2020.*

3 *"Data centres & networks." IEA, 2022.*

4 *"Electricity demand graph."Our World In Data, accessed October 2022.*

The energy usage of the world's datacenters is usually estimated as between 200 and 250 TWh—even excluding cryptocurrency mining. This is a similar scale to the usage of a medium-sized country, such as Brazil or the U.K. The 200 TWh figure does not include the energy requirements of the network traffic running in and out of the datacenters. With network traffic included in the calculation, the electricity usage of data and datacenters is approximately double the electricity usage of South Korea.

Although electricity can be low-carbon, it is far from guaranteed. Most electricity is still produced by burning of fossil fuels; in 2021, 62% of electricity was generated by burning fossil fuels.

Sustainable software engineering

Green software engineering principles give a broad framework for thinking about sustainability of software. The carbon impact of software has three parts:

- Hosting
- The software itself
- User behavior

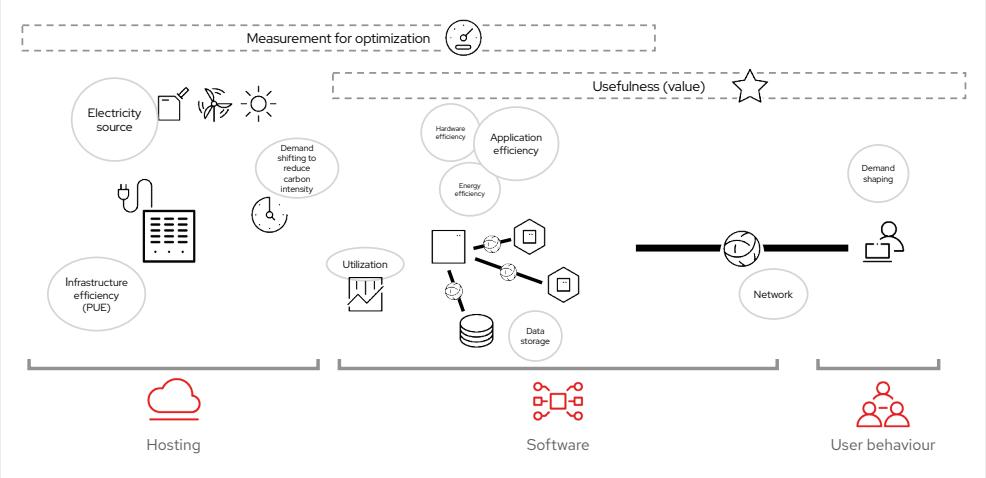


Figure 3. Principles of green software engineering overview [5]

Hosting

How and where an application is run determines a large part of its carbon footprint.

Electricity source

Not all electricity is created equal. Some electricity is generated by burning fossil fuels, such as coal and gas, while some comes from renewable sources such as hydro, solar, and wind. What kind of electricity you get depends on the hosting region, the weather, and the time of day. Different areas

5 *Adapted from "Principles of green software engineering." Principles.Green, accessed October 2022.*

have different energy mixes, depending on the local infrastructure. For example, in Virginia (home to many datacenters), the mix is heavily dominated by coal. In many Nordic countries, renewable energy such as wind and solar power are more available.

Many cloud providers now publish information about the carbon profile of their various regions, which is helpful in making carbon-conscious choices.

Infrastructure efficiency

Another factor which influences the carbon footprint of hosting is the infrastructure efficiency. Efficiency mostly matters when comparing equivalent energy mixes as an inefficient center running on carbon-free electricity emits less carbon than an efficient one running on fossil fuels.

The good news is that the efficiency of datacenters has improved considerably over the past two decades. Innovations in cooling technologies, economies of scale, and reduction of waste have allowed some providers to achieve near-ideal PUEs. The PUE measures how many units of energy have to be put into a facility per unit of computation. An ideal PUE is 1.0—a datacenter with no overheads at all. A normal enterprise datacenter might have a PUE of 1.6-1.8, whereas some hyperscalers report PUEs of 1.1.

The software itself

Application efficiency

For most IT professionals, application efficiency is the one aspect of software-associated carbon emissions over which they have the most control. Applications should be optimized to use as little energy as possible, and also to run on low-specification hardware.

Data storage

Data should not be overlooked when considering efficiency. Storing data has an environmental cost, and so storing more data than needed can adversely affect footprint.

Network

Like data, network traffic is a less visible element of carbon footprint. A network is itself a large collection of computers, such as switches, routers, and servers. Each of these devices consumes energy and has embodied carbon from its manufacture. Aim to reduce the volume of network traffic into and out of systems. You should also try to reduce the distance data travels.

User behavior

How active a software system is depends on how much its users use it. Application authors should consider helping users to make carbon-aware choices. This could be as simple as a toggle to enable dark mode, or as sophisticated as a nudge to reward demand during off-peak hours. Request batching, eco-modes, and bandwidth downgrading are all levers which can empower users to drive carbon reduction.

Cross-cutting considerations

Usefulness

Have you ever streamed a music video when you only wanted to listen to the song? Or let a CI/CD job run for hours, even though you did not actually care about the result? Software can do extraordinarily useful things, but it can also do entirely useless things, at scale.

Measurement

You cannot optimize what you cannot measure. Just as an organization might monitor response times and error rates, it should monitor the carbon footprint of its software estate. Measuring carbon has some complexity, so we will return to the subject later.

Decarbonisation solutions

No-regrets solutions

Some carbon reduction techniques involve discomfort, or doing without, but many do not. These are win-wins, or what Project Drawdown calls no-regrets solutions. The second win is known as a co-benefit.

As a simple example, one cloud provider recently reported that running workloads in its Montreal, Canada datacenter was 88% less carbon-intensive than running the same workload in London, England. This is not too surprising, but what may be surprising is that hosting in Montreal was also 15% cheaper. If trans-Atlantic latency is a concern, hosting in Finland used 43% less carbon than London, and was also 15% cheaper. Unless there are other constraints, such as latency or data locality, it is a win-win to host in a low-cost, low-carbon region.

Although this is a specific scenario, the same double-win effect can be observed with many carbon solutions. As well as reducing carbon, they often reduce cost or improve user experience or carry other co-benefits.

Turning off zombie servers

A surprising proportion of datacenter electricity goes to applications which are not actually in use. Known as "zombie servers," these applications stopped being useful and should have been permanently decommissioned, but no one got around to it. Estimates suggest around 30% of running servers are in this abandoned state, doing no useful work.

LightSwitchOps

Ideally, no IT system should be "always on" unless it is actually needed. If an application is not being used by anyone, it cannot be providing value. Eliminating these idle applications could significantly reduce greenhouse gas emissions—with no downside. In fact, there is a considerable upside, cost savings. Even using a simple script to turn servers off at night and turn them on in the morning can give a cost savings of around 30%.

"One estimate concluded US$26.6 Billion of public cloud spend was wasted on idle workloads in 2021."

Kathy Stalcup
"Overprovisioning & always-on resources lead to $26.6 billion in public cloud waste expected in 2021," *Business 2 Community*, 21 Jan. 2021

Although no workload should be always-on, in practice many are. Businesses may leave applications running 24/7 even if they only need them during business hours, because they do not have the confidence to scale them up and down. What's behind this lack of confidence? Often, insufficient elasticity is the underlying problem. Elasticity is a measure of how simple it is to scale up and down. Think about a light switch. With a light switch, you turn the lights off when you leave the room, and you turn them on again when you come back. The way a light turns on and off with almost no friction reflects the high elasticity of a light bulb.

LightSwitchOps describes the automation and cloud-native architectures to support frequent scaling up and down. One barrier to the wider adoption of LightSwitchOps is limited tooling to detect situations of low- or high- load and scale appropriately. However, the space is developing rapidly, with

a range of options to handle auto-scaling instances and even clusters. We expect new standards and solutions to continue to appear in this area. Adoption of DevOps techniques, or perhaps GitOps workflows, may also help organizations build confidence to turn applications off temporarily.

LightSwitchOps can handle both the problem of servers, which should be scaled down when they are not used, and forgotten servers which are never used. If it is less complicated to spin systems up and down, there is less temptation to leave systems up past the point when they stop ever being used.

LightSwitchOps and Java

Traditionally, Java has not been very much like a light switch. Compared to languages like Go and Rust, a Java application has a long warm-up time. This is fine for a long-lived application, but a problem for short-lived applications or serverless scenarios.

The good news for the Java community is that it is changing; some Java start-up times are now comparable to light switches, and in some cases even faster. A simple REST application on Quarkus native starts up (and gives the first response) six times faster than an industry-leading light-emitting diode (LED) light bulb.

A simple native Quarkus application will turn on faster than an LED light bulb.

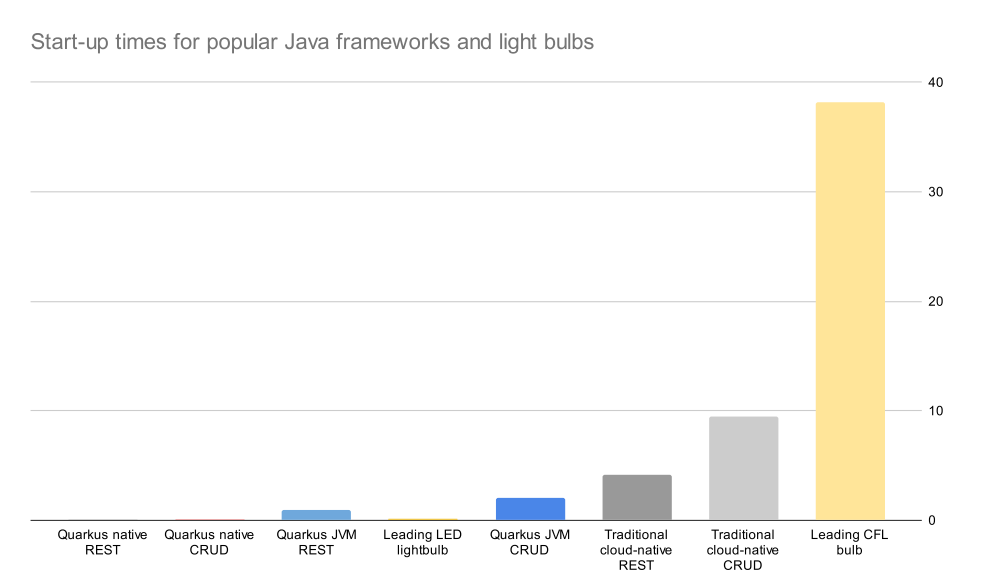


Figure 4. This figure shows the warm-up times for frameworks and lightbulbs. For the frameworks, the time is the boot up time plus the first response time. For the lightbulbs, it shows time to full brightness.

The Quarkus native and LED times are too small to see when they are mixed in with the longer times. This figure shows just those times.

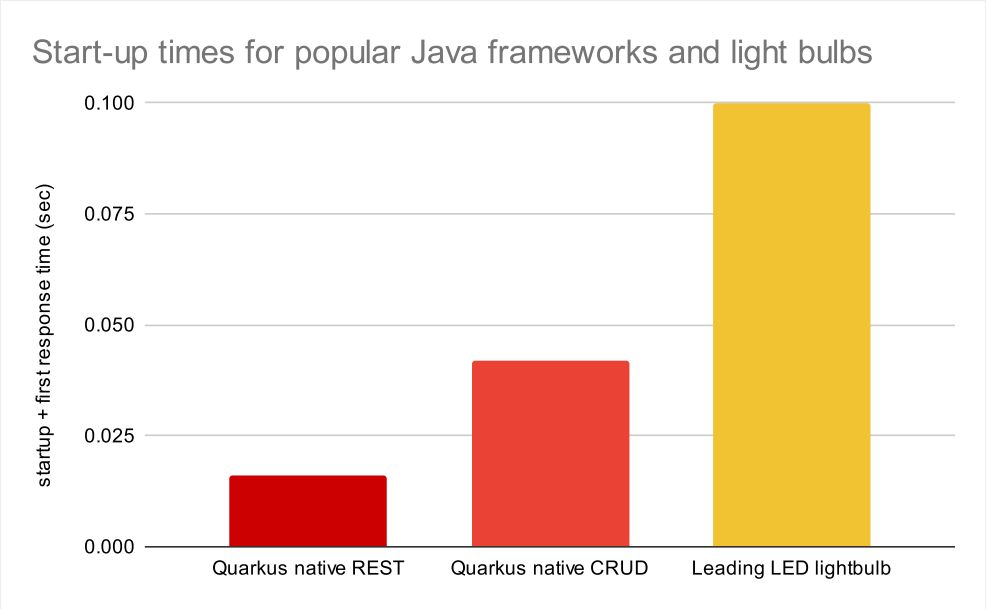


Figure 5. Start up and first response times [6] [7]

Efficiency

Efficiency is an excellent no-regrets solution. It saves money, it saves carbon, and there are often tertiary benefits, too, like improved developer experience, more harmonious user experience (UX), better accessibility, and higher user conversion rates.

How Quarkus reduces carbon

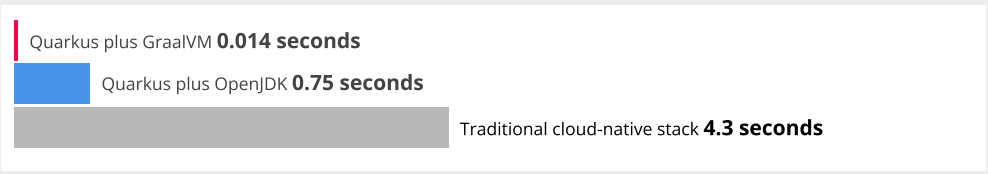
This is where Quarkus comes in. Quarkus is a Kubernetes-native Java stack built on proven Java libraries and standards and designed for containers and cloud deployments. A Quarkus application can run on the Java virtual machine (JVM), or it can be compiled into a native binary using GraalVM.

As discussed earlier, Quarkus in native mode starts in less time—comparable to a light bulb. Even when running on the JVM, Quarkus applications start fast. A simple Quarkus REST application starts in around a quarter of the time of traditional cloud-native stacks.

6 *"Power up delay when LED lights are turned on–Is this normal?"* LampHQ, accessed October 2022.

7 *"Supersonic/subatomic Java."* Quarkus, accessed October 2022.

Quarkus describes itself as supersonic subatomic Java, which is a way of saying that it's really fast, and it's really light.



For a REST + CRUD application, Quarkus's start-up time results are similarly small:

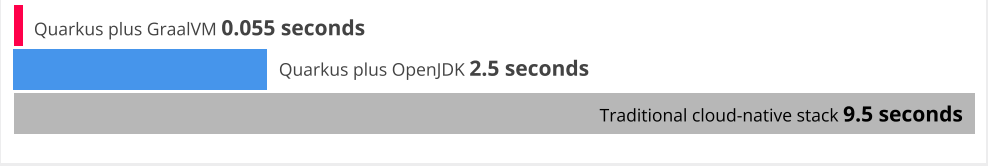


Figure 6.Quarkus application start-up time comparison [8]

In JVM mode, the memory footprint of the Quarkus application was just over half the memory of a traditional framework, and in native mode, it was around a tenth of the memory.

REST application

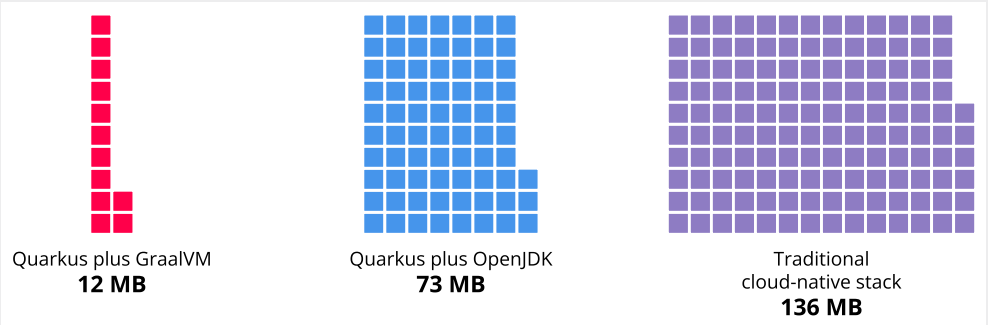


Figure 7. Memory usage of a REST application, using Quarkus or traditional cloud native runtime [9]

Do these memory footprint and startup-time figures translate into reduced energy usage over the lifetime of a typical app? The short answer is "yes".

Capacity test

Quarkus performance engineers did some experiments, using RAPL to monitor power consumption. The application was a basic create, read, update and delete (CRUD) application that queries a database via a REST interface. The application is started and warmed up. A load generator (wrk2) is then used to send requests at a fixed arrival rate (injection rate). While the application is under load

8 *"Quarkus runtime performance." Quarkus blog, 7 July 2019.*

9 *"Quarkus runtime performance." Quarkus blog, 7 July 2019.*

power consumption of the CPU and DRAM was measured using running average power limit (RAPL). The machine was a two CPU machine, with 16 cores per CPU. The application's CPU affinity was set to four specific cores on one CPU, and the application was the only process running on those cores. The heap was not pinned, and could go up to 12G.

The team found that the application running on the traditional cloud-native framework could not support as many incoming requests as Quarkus (which is why the lines are shorter in the graph).

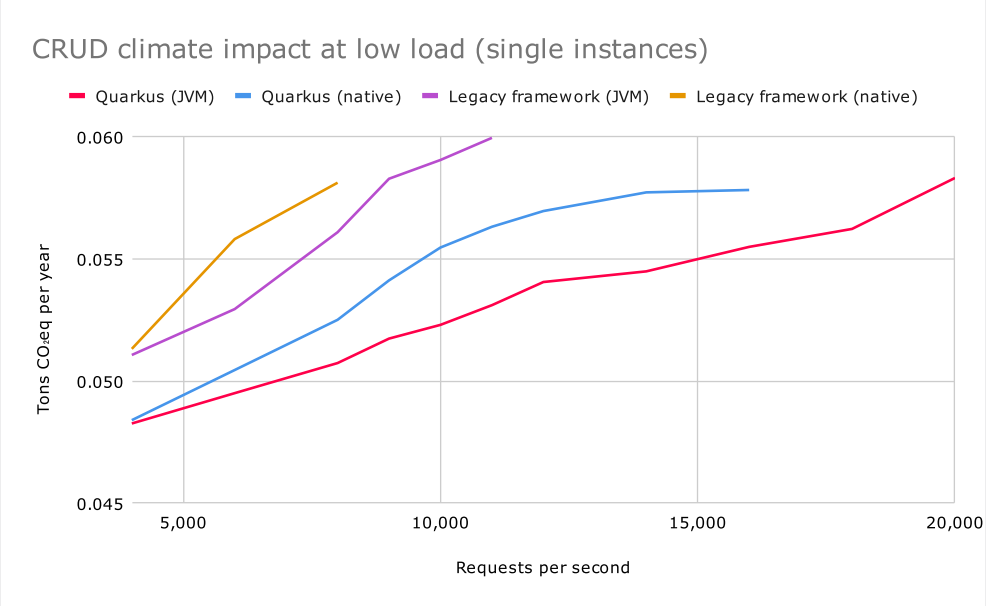


Figure 8. Carbon footprint of a single-instance CRUD application as a function of load

Assumptions:

- The CO2-eq figures are based on the U.S. energy mix.

Note that the y-axis of this graph does not start at zero. The system had a baseline power consumption at all loads.

For a given number of requests, Quarkus accounted for the least energy and CO2-eq. Quarkus native was next-most efficient, then the traditional cloud-native framework, and lastly the native version of that framework. (See figure 9).

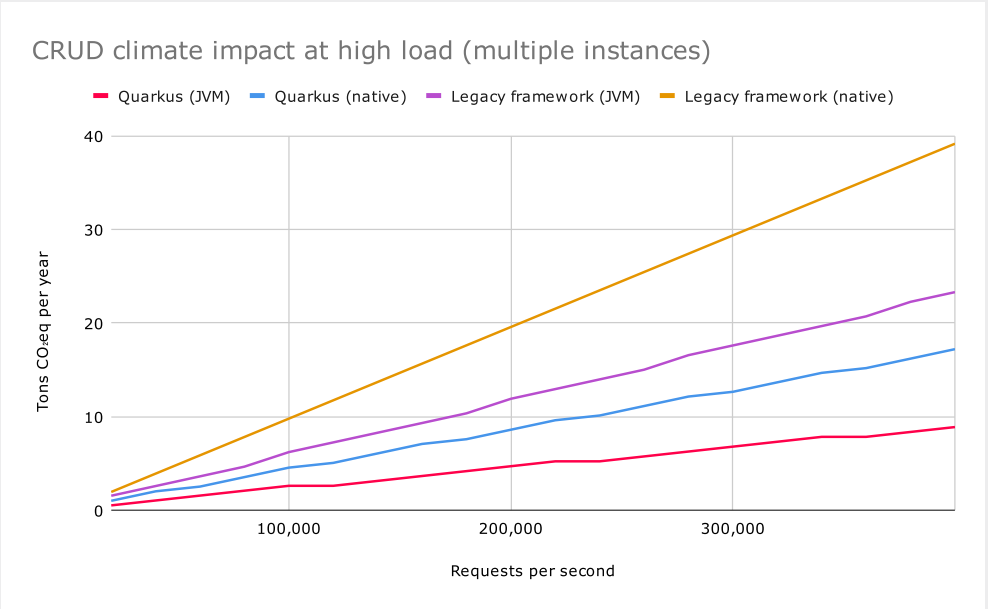


Figure 9. Carbon footprint of a multi-instance CRUD application as a function of load

Summary

In these experiments, the applications running on Quarkus were responsible for significantly less energy usage—and therefore, carbon emissions—than the same application running on a traditional cloud-native framework.

For both Quarkus and the traditional framework, more energy was needed to handle the load in native mode than running on the JVM. This may not be what you expected. Read further for some considerations about why (and when) native might consume more energy.

These trends hold at both low loads, when a single application instance could be used, and at high load, when multiple instances are required.

Density test

What happens in the more resource-constrained environments of the cloud? Because of its small footprint, the density of Quarkus deployments can be much higher than with traditional cloud-native stacks. What kind of carbon savings can this give?

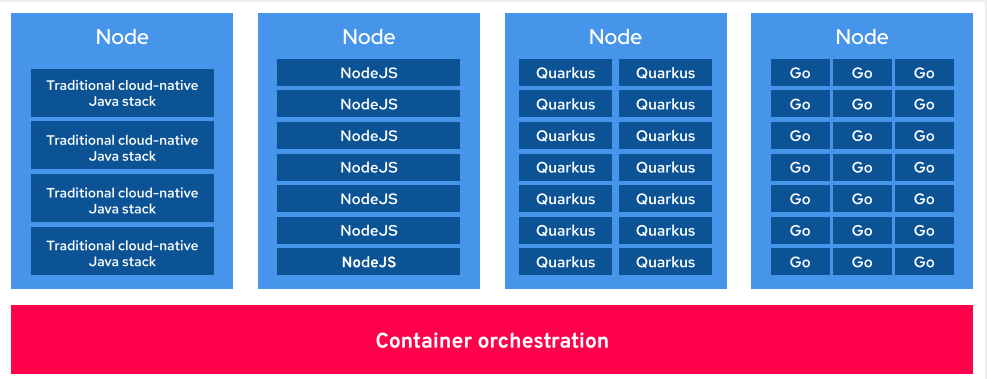


Figure 10. Container orchestration diagram

To find out, the Quarkus team deployed an application to the cloud, and loaded it with 800 requests per minute over 20 days. The experiment was attempted with several instance sizes and application frameworks. When the application was running a popular legacy cloud-native framework, the minimum instance which could handle the load was an AWS t2.medium instance with two CPUs and 4GB memory. When running on Quarkus, a much smaller instance could be provisioned: a t2.micro instance with 1 CPU and 2GB memory was sufficient.

Table 1. Comparing traditional framework on JVM to Quarkus

Framework	Instance Type	Price per hour	CO_2-eq per hour
Legacy framework on JVM	t2.medium (2 vCPU, 4GB)	$33.40	5112 g
Quarkus on JVM	t2.micro (1 vCPU, 1GB)	$8.40	2376 g
Quarkus native	t2.micro (1 vCPU, 1GB)	$8.40	2376 g

On average, Quarkus used 2-3 times less carbon than the legacy framework (See figure 11).

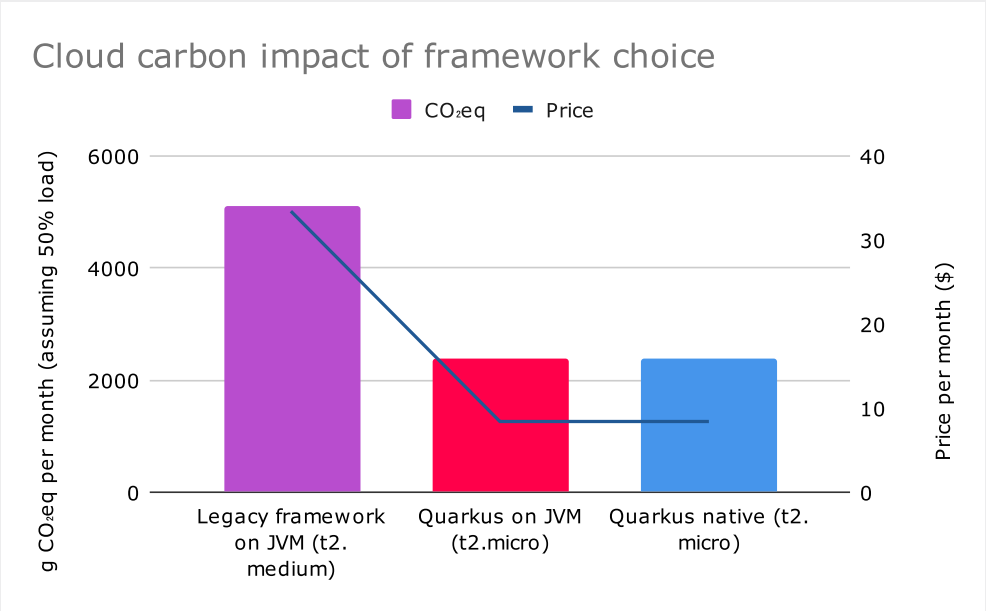


Figure 11. The effect of framework selection [10]

Assumptions:

- Figures assume us-east-1 datacenter
- Figures assume 50% load
- Measurements were made between May and August 2021. For this reason, it was not possible to try a native version of the legacy framework.

Data sources:

- Performance lab measurements

The results of the density test are consistent with the capacity test. To run the same workload, Quarkus required approximately two times less carbon than the traditional framework.

Running on a smaller instance means less cost, but also less carbon. The carbon from running an application comes from two sources: the carbon of running the workload, and also the embodied carbon in the hardware itself. Running in a low-carbon datacenter—such as Stockholm, Sweden—will reduce the runtime emissions, but not the embodied carbon. This means it is important to consider hardware requirements of an application when thinking about greenness.

If you know what instance you are running and in which region, you can get an estimate of the carbon per hour using publicly available datasets, such as the Teads carbon dataset which was incorporated into the cloud carbon footprint (CCF) tool. The Teads dataset only has data for four predefined load levels, so here we have assumed a uniform 50% load.

10 *"Carbon footprint estimator for AWS instances." Teads Engineering, accessed October 2022.*

One difference between the two sets of results is that in the constrained environment, with a low-spec system and limited load, native mode Quarkus did not cause more carbon emissions than the JVM mode. In fact, the native application likely used slightly fewer resources than JVM. It is not possible to reflect this in the results table because of the coarse granularity of the load to carbon conversions.

Architectural choices and carbon

Native Quarkus or JVM Quarkus?

Why do the results of the density test and capacity test give different answers to the question "is native mode more carbon-efficient than running on the JVM?" In the density test, the workload is fixed (800 transactions/s) and the goal of the experiment was to size machines as small as they could go. Native could go smaller than JVM, but because of the rather coarse granularity of the machine specs, that is not reflected in the figures. You cannot provision a machine with 427mb of RAM.

In the scalability test, the machine had a generous specification (although the application was limited to four cores). As much load was driven into the system as possible, and then power consumption was measured. If a single instance could not handle all the load, more instances would be added. In this scenario, carbon efficiency turns out to be closely correlated with framework throughput. Application throughput is slightly higher running on the JVM than in native mode.

You may find this counter-intuitive, because usually fast things use more energy, not less energy. For software, the mental model needs to be different. Faster software does more (in a unit of time) with the same system resources. This means it is more efficient and uses less energy.

Use case for native (considering carbon efficiency)

- Low workload (throughput is not the bottleneck)
- Resource-constrained or old hardware (especially memory)
- High redeploy rate (applications never get warmed up before being spun down)
- Inconsistent workload (you need to deploy and undeploy instances elastically)
- Serverless

Use case for JVM (considering carbon efficiency)

- High workload (you need lots of throughput)
- Long-lived processes (the rapid start of native does not save you much over the lifetime)

Remember, though, that native and JVM are not a matter of choice. Since Quarkus applications can run in either mode just by adding a compile flag, there is a great deal of flexibility. Energy mixes and load can vary hour-by-hour, so a hybrid model will often be the winner.

The decision between native and JVM mode can be revisited many times, at almost no cost. Contrast this, for example, with trying to save carbon by coding in C++. That decision would have to be made upfront and would be extraordinarily difficult to reverse.

It is unlikely for a production system to know exactly, in advance, what kind of load it will receive—unless it is a specialized system not exposed to external traffic. Often, whoever is responsible for such a system's stability will be nervous enough about the unknowability of load to require some generous

additional slack capacity to be ready to go. For a user to perceive a system as instantaneous, it needs to respond to requests in around 0.1 seconds. This means extra instances should serve requests within this window, or close to it.

If backup servers are based on a JVM application, they will need to be booted well in advance. Ideally, they also would be booted up long enough in advance to warm up the application, to allow JIT optimisations to start running.

If an application is native-based, no warmup is necessary. This simplifies things; instances might still need to be booted with some advance notice, but less so. Being able to compensate for load spikes in just seconds (once infrastructure spin-up time is included) removes a lot of the stress of capacity planning. The smaller footprint of each instance also helps with better granularity.

An ideal system might combine some JVM based nodes (efficient big irons for the main workload) with some native-image nodes to improve elasticity and maybe occasional support during rolling upgrades. Quarkus's ability to compile the same application in the two different modes is a strong asset for modern capacity planning and greener software.

Reactive programming or imperative programming?

Is reactive more carbon efficient than traditional programming models? It is complicated.

The reactive principles arise from the need to build more elastic, resilient, and efficient systems. They often rely on an execution model using fewer threads, with high mechanical sympathy, so in principle, they use resources more effectively.

However, you often use a modern reactive programming library to build reactive applications, which tend to have more allocation. This is because, while the application uses fewer threads, you need these threads to collaborate. Thus, you need a back pressure protocol to avoid overloading parts of the application. Currently, the reactive programming libraries are not especially efficient regarding CPU, memory, and carbon.

Countering this is the fact that correctly implemented back pressure will pace the load of your system, avoiding bottlenecks and other expensive situations. Loads which, despite being moderate overall, are volatile or "bursty" can turn out to be the main reason for bad performance and poor efficiency of the underlying application. For example, bursty loads trigger excessive kernel parking and unparking, which is inefficient.. A general heuristic for avoiding this situation is to give the application fewer resources and keep the system busy enough that idle periods are avoided during the burst. The aim is to avoid kernel parking and unparking during a small lull in the middle of a burst.

What does that mean for your choice of programming model? Like many architectural decisions, it depends. The efficiency of reactive will vary depending on the application and execution context.

Measuring carbon

Regular measurement and monitoring of carbon footprint (as in its impact) is one of the green software principles. It's important to measure carbon, so that you know what to change to reduce it. However, accurate carbon measurement can be difficult.

Climate change is caused by the accumulation of greenhouse gases in the atmosphere. Greenhouse gases include carbon dioxide (CO_2), methane, nitrous oxide, and hydrofluorocarbon refrigerants. The most common greenhouse gas is carbon dioxide (CO_2), but some of the other gases have an even

more potent warming effect. For ease of comparison, all greenhouse gas measurements are normalized to carbon dioxide equivalent (CO_2eq). As a shorthand, carbon is often used to refer to all greenhouse gases.

At a high level, the carbon footprint of software can be calculated by:

- Measuring the energy usage of the workload—how big is the computer, and hard is it working?
- Working out the carbon intensity of the electricity—was the electricity generated by burning coal, or natural gas, or cleanly, from hydro dams, solar, or wind?
- Adding in the carbon footprint of network traffic.
- Adding in the embodied carbon of the hardware.

This simplified set of steps omits many indirect carbon contributions from the software life cycle. These include activities such as transportation costs of the hardware, construction costs of the datacenter, and other overheads.

Carbon footprint is usually divided into three scopes:

- **Scope 1** is direct emissions. It counts the direct effect of fuel consumption (or other GHG-emitting activities). It is counted by whoever owns the fuel when it burns. For example, it would cover fuel use by a car owner, or by an electricity generator..
- **Scope 2** is indirect emissions related to energy use (electricity and heating). It's a measure of electricity consumed.
- **Scope 3** covers emissions across the value chain. This would include the manufacturing emissions from hardware, buildings, and other infrastructures and transportation emissions across the supply chain. It also includes upstream effects from consumption of a business's service.

You can see these calculations soon become complex. Direct measurement is often impossible, and so carbon calculations are almost always a combination of measurement and estimation (or modeling). Measurement is accurate, but it can be intrusive and is often impossible.

Kepler

Kubernetes-based efficient power level exporter (Kepler) is a collaboration between Red Hat and IBM Research. It uses a lightweight and efficient probe written in eBPF to collect system and process information, including CPU runtime, performance counters, and correlates with RAPL readings to estimate power consumption by pods. The estimations are processed through Prometheus and can be visualized on Grafana and other cloud-native consoles.

Measurement for optimisation

When reporting carbon, comprehensive calculations are difficult, but important. When optimizing carbon, it is often unnecessary to do a fully accurate calculation. What matters is the relative carbon of two different scenarios, and this may be simpler to calculate. If you can figure out what the slope looks like, you can optimize by going down the slope, even if you do not have absolute values.

Load and other performance metrics as a proxy for carbon emissions

Although it is not as accurate as a proper measurement, our performance and carbon measurements show a loose—but useful—correlation between performance metrics and energy consumption. For example, if a change to an application means it can handle the same volume of requests with the

same memory footprint and reduced CPU load, it is almost certainly using less energy (assuming the change was not to offload processing elsewhere). Similarly, reductions in memory footprint or network traffic, assuming all else stays the same, are likely to be improvements in carbon emissions.

Cost as a proxy for carbon emissions

Another potentially useful proxy for carbon emissions is cost. The cost for cloud computing is (usually) quite visible, and it can be a useful heuristic. There are, however, limitations:

- Coal fired electricity can be cheap in the short term. When using cost as a proxy, we need to compare apples with apples.
- Even if the overall trends go in the same direction, it is not linear. For example, in the density experiments discussed earlier, the price difference between Quarkus and the traditional cloud-native framework was US$26.08 vs US$9.15, and the carbon difference was 7.1g vs 3.3g. The price difference was greater than the carbon difference (a factor of three vs a factor of two). This ratio is specific to the us-east-1 region. In other regions with cleaner energy, large changes in cost would be associated with even smaller carbon differences.

The model can be made more concrete, using some calibration. For example, once a breakdown of spend categories has been done, a business can work out a ratio between cloud costs and carbon similar to emitting $x metric tonnes of carbon for every $y million of cloud spend. This is known as economic input-output life cycle assessment. Industry-standard emissions factors are now becoming available for various categories of cloud spend. The model is necessarily coarse, but it can be convenient, simply because the financial data is so widely available and calculations are fast.

With those cautions, cost reductions can be a reasonable guide to carbon reductions, in situations where other factors stay the same.

Summary

Quarkus can reduce carbon emissions in a number of ways. Quarkus itself is a highly efficient runtime, and applications running on Quarkus will use fewer resources, and therefore, emit less carbon. This carbon reduction is automatic because no effort is required beyond the effort (if any) of migrating to Quarkus. It is passive, because you do not need to do anything except use Quarkus.

You can go further and get extra carbon savings. Quarkus enables more carbon-efficient architectures. For example, in many circumstances, the reactive option provided by Quarkus can yield carbon savings. At a hardware level, you may be able to change what you provision, and run your workloads on smaller machines. This saves the runtime energy, and it also saves the embodied carbon in the hardware.

Quarkus also supports elasticity, which means servers can be kept completely off until needed. In native mode, application instances come up near-instantaneously and so can handle high-bursty traffic with no degradation of quality.

For many scenarios, a hybrid model combining native and JVM instances may provide the optimum combination of high throughput, high carbon-efficiency at load, and low energy utilization at idle.

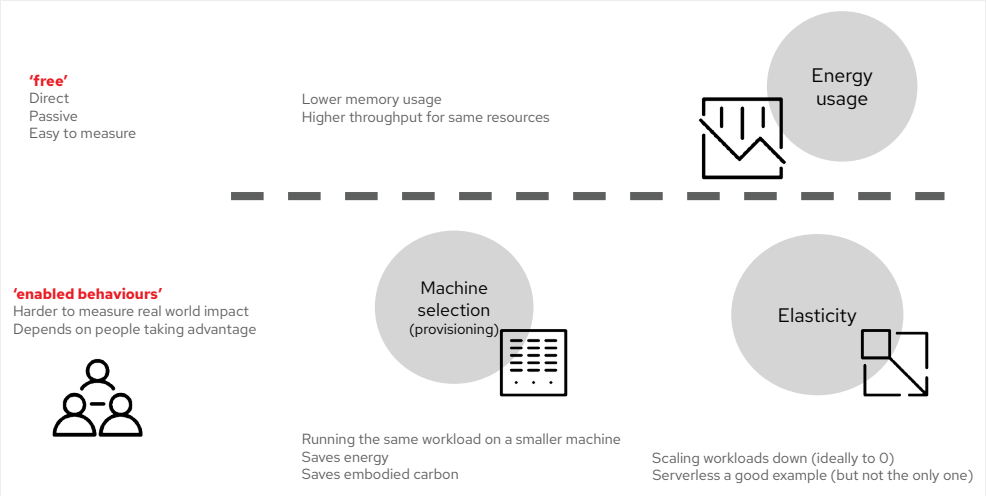


Figure 12. Ways in which Quarkus can help reduce carbon footprint

Customer testimonials

Here are a few examples of Red Hat customers who have reduced resource usage by switching to Quarkus. These stories tell how Quarkus's resource savings translate from the performance lab to production.

Lufthansa

"With Quarkus, we could run 3 times denser deployments without sacrificing availability and response times of services."

Thorsten Pohl
Product Owner
Automation & Platform Architect
at Digital Product Division, AVIATAR

Lufthansa Technik runs a digital platform called AVIATAR. To scale software development, they migrated to a microservices architecture. This improved developer productivity, but they noticed a negative effect on their resource usage. The AVIATAR application was composed of 100 services, each running in triplicate for high availability. Some services required significant CPU and memory; multiplied by three hundred service instances, the overall resource consumption of the microservices architecture was far too large. Lufthansa is now migrating services to Quarkus. The company is using a mix of native and JVM, depending on the service. Some services are even switching between the two.

Quarkus has also helped them to introduce serverless into the architecture, for further resource savings. The native applications come up rapidly enough Lufthansa can leave them switched off without degradation of service.

What has the outcome of the migration to Quarkus been? Lufthansa has observed the new services are using a third of the resources than previously. For example, the Quarkus version of the old 0.5 core 1 GB service requires only 200 millicores plus 200–400 MB of RAM per instance.

Vodafone

Vodafone is migrating part of their architecture from Spring Boot to Quarkus, to lower resource usage. In the past, some microservices required 1 GB of RAM when using Spring Boot. For production, they can now deploy a Quarkus microservice with 512 MB of RAM. "For 80 microservices, this is a big savings!" Christos Sotiriou, DXL technical lead at Vodafone Greece, emphasized. He added, "What Quarkus offers by default (without trying to optimize it) is 50%–60% more lightweight (in JVM mode)

than what Spring offers after optimizations (taking care of dependencies, playing with JVM options, etc)." Although Vodafone's primary motivation was cost, the lowered resource usage will also have green benefits.

Decathlon

When designing their new VCStream messaging platform, Decathlon chose Quarkus partly for the ecological benefits of its low resource usage. The VCStream system has achieved impressive efficiency. Decathlon measured 1 million messages per minute throughput, per CPU per GB of memory. This was a greenfield application, implemented only once, so there is no comparison to a previous implementation.

Learn more

For more details about Red Hat® build of Quarkus, see the Quarkus overview.

facebook.com/redhatinc
@RedHat
linkedin.com/company/red-hat

North America
1 888 REDHAT1

Europe, Middle East, and Africa
00800 7334 2835
europe@redhat.com

Asia Pacific
+65 6490 4200
apac@redhat.com

Latin America
+54 11 4329 7300
info-latam@redhat.com

redhat.com
#F32147_1122