Green Coding

Climate change is one of the greatest challenges that will face humanity in the coming decades; information and communication companies can make a difference with GreenCoding.

Motivation

It is the 20th January 2021. The new president of the United States, Joseph R Biden Jr, has just been inaugurated and now wants to take action on climate change, promising $1.7 trillion of investments in clean energy and net-zero emissions in the United States by 2050.

Meanwhile, across the pond, the European Commission has committed to providing €100 billion in investments for the transition to climate neutrality in the same period to achieve policy objectives and commitments such as the Paris Climate Agreement at COP 21.

Therefore, it is evident that reducing one's CO_2 emissions will become an even higher priority for companies and associations globally.

Table of content

Furthermore, the growth in investment in renewable energy sources and green technology such as electric cars illustrate that whilst these areas are significantly gaining traction, they remain in their infancy the next years; especially as in 2019, only 11% of the world's primary power was derived from renewable sources. Therefore, the priority should be on demonstrating reduced emissions through innovating business processes and value chain by taking a lifecycle assessment approach which ensures that overall energy and resource is reduced, not merely lowering CO_2 emissions. Analysing core processes across the whole value chain means that over time, by implementing incremental changes, you will be able to make a substantial reduction not only in emissions and resource usage, but further make substantial gains in improving overall efficiency.

Currently, most companies only focus on their direct emissions, known as Scope 1 and 2 emissions according to the GHG (Greenhouse Gas) Protocol. These are mainly caused by certain processes of goods production – e.g., refrigeration and heating. But for many organisations the biggest impact is derived from indirect (Scope 3) emissions, also referred to as value chain emissions such as those linked to actually using products.[1]

Scope 3 emissions are particularly relevant in the information and technology sector, since emissions stemming from development only account for a small share. Using Microsoft as an example; from a total of 16 million metric tons of carbon emissions in 2020, about 75% fall into scope 3.[2] In general, programming is always about efficiency of effort; but do developers really take energy efficiency into account when coding?

Recent studies show that electricity demand in information and communication areas currently account for between 5% and 9% of global electricity demand, and forecasts predict this number could rise to as much as 21% by 2030.[3,4]

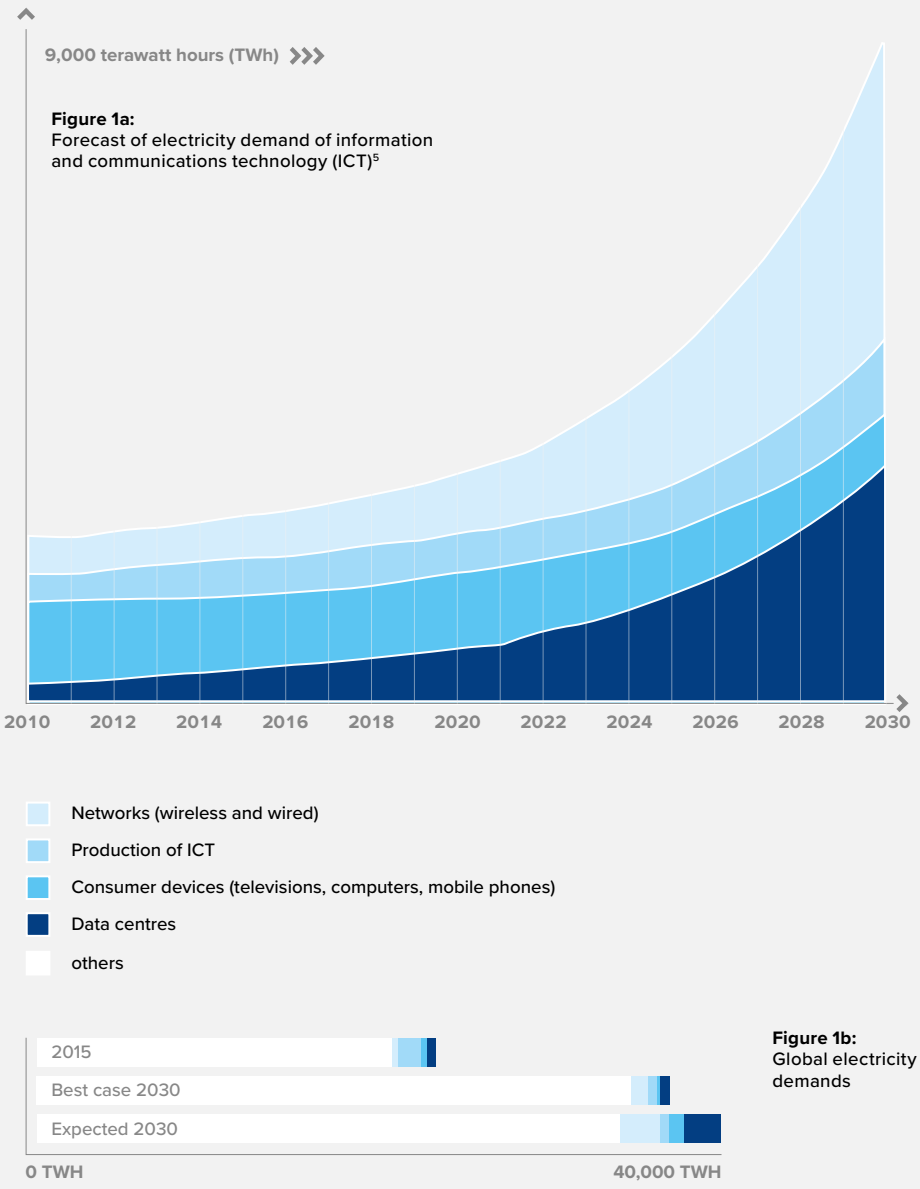


Figure 1a:
Forecast of electricity demand of information and communications technology (ICT)[5]

Figure 1b:
Global electricity demands

[1] WWF Germany, "Overcoming barriers for corporate scope 3 action in the supply chain", https://www.wwf.de/fileadmin/fm-wwf/Publikationen-PDF/WWF-Overcoming-barriers-for-corporate-scope-3.pdf, December 2020

[2] Microsoft "Microsoft will be carbon negative by 2030", https://blogs.microsoft.com/blog/2020/01/16/microsoft-will-be-carbon-negative-by-2030/, September 2020

[3] Huawei Technologies Sweden AB, "On Global Electricity Usage of Communication Technology: Trends to 2030": https://www.mdpi.com/2078-1547/6/1/117, April 2015

[4] Enerdata, "Between 10 and 20% of electricity consumption from the ICT sector in 2030?": https://www.enerdata.net/publications/executive-briefing/between-10-and-20-electricity-consumption-ict-sector-2030.html, August 2018

[5] Springer Nature, "How to stop data centres from gobbling up the world's electricity": https://www.nature.com/articles/d41586-018-06610-y, September 2018

Why companies need to be aware of this topic – and why programmers should think twice about coding

Public awareness of the possibilities offered by sustainable software development is minimal, although pioneers like Alex Russell and Jeremy Wagner have been trying to change this situation for some time.[6,7] To make a difference, it will be necessary to raise awareness in all areas and amongst all stakeholders – business, delivery, consumers and creators.

From a business or management perspective, this low awareness of software energy consumption is mainly driven by one factor; that development and operating budgets are mostly 'decoupled'. As a result, there is an obvious conflict of interest. On one hand, more development time needs to be invested to carry out efficiency testing, but on the other hand, operating costs rise when you work less efficiently. Managers therefore need to focus on sustainability as the ideal outcome. They should move away from the traditional priority, which was purely about delivery performance and cost reduction, and focus instead on an across-the-board priority - optimising software end-to-end.

Obviously, management is just the first layer in this suggested change of mindset. Software analysts, architects and engineers also need to be engaged if companies wish to embark on this journey, based on the premise that computers are merely machines, like any other. Energy efficiency depends on the software that machines run. All code creates a carbon footprint, so the onus is on all of us to ensure this footprint is as small as possible.

If we think about this issue in terms of ultimate performance on a global scale – with cloud providers continuously operating infrastructure servers, working alongside other providers and companies – there is considerable potential to save energy. It should, however, be noted that this not only applies to conventional applications such as operating systems, office technology or server applications. Scaled up to hundreds, thousands or even millions of devices (desktops, smartphones, tablets...), every single piece of code can make an important contribution to reducing one's energy consumption, thereby helping to reduce overall CO_2 emissions.

[6] "Alex Russell – The Mobile Web: MIA": https://vimeo.com/364402896, October 2019
[7] "Responsible JavaScript" by Jeremy Wagner: https://speaking.jeremy.codes/Vci5ad/responsible-javascript, October 2019

The scope and potential of GreenCoding

This is where so-called GreenCoding comes in. The idea is to program, deliver and run software in a much more environmentally friendly way.

Sphexishness [sfɛksɪʃnəs] is a term coined by Douglas Hofstadter for being caught in a rut of automatic thought

A key prerequisite for GreenCoding is to fundamentally think again. This involves taking a holistic approach to issues or business problems and wherever possible avoiding ingrained, automatic thinking – or 'sphexishness'.

GreenCoding starts with the planning of a project when initial requirements are being analysed. At this stage, the key priority is to select a suitable platform for the development process. Studies have found that some programming languages already have a major impact on energy efficiency and speed; allowing programmers to choose an appropriate programming language can make a massive difference when it comes to energy savings and performance. It should be noted that this is just an example, however, depending on the project, there could be decisions that have an even greater impact.

Ultimately, GreenCoding will mean adding a new question to the design process. Teams need to question if there is a better way to deliver the desired benefit with the least possible amount of energy. Answering this question may have a huge impact on design. For example, it could lead to a 'serverless' approach in order to optimise infrastructure, or you might decide to adapt the user experience to minimise the time invested by the end users of the software.

TOTAL

ENERGY		TIME		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

Figure 2:
The energy, time and memory required by different languages for a benchmark problem[8]

[8] Energy efficiency by programming language: https://greenlab.di.uminho.pt/wp-content/uploads/2017/10/sleFinal.pdf, October 2017

There are three important pillars when it comes to writing software:

These three pillars affect code in different ways, and each needs to be dealt with separately. However, the first task is to lay the foundations on which these pillars depend and establish a high-level overview before implementing the delivery process mechanisms.

"WHAT"

is generated (the code itself). Is it efficient in terms of delivered benefit or energy invested?

A GREENER LOGIC

"HOW"

it is generated. Is the software development life cycle efficient? Could the same code be generated with less energy?

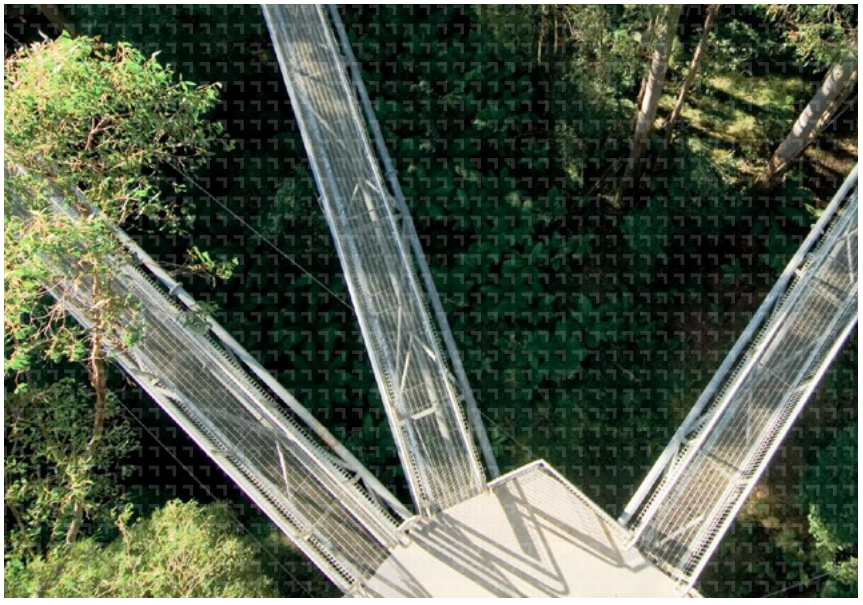
A GREENER METHODOLOGY

"WHERE"

the software is operated (the final platform running the code). Is it consuming the minimum energy required to run the generated code?

A GREENER PLATFORM

ARCHITECTURE

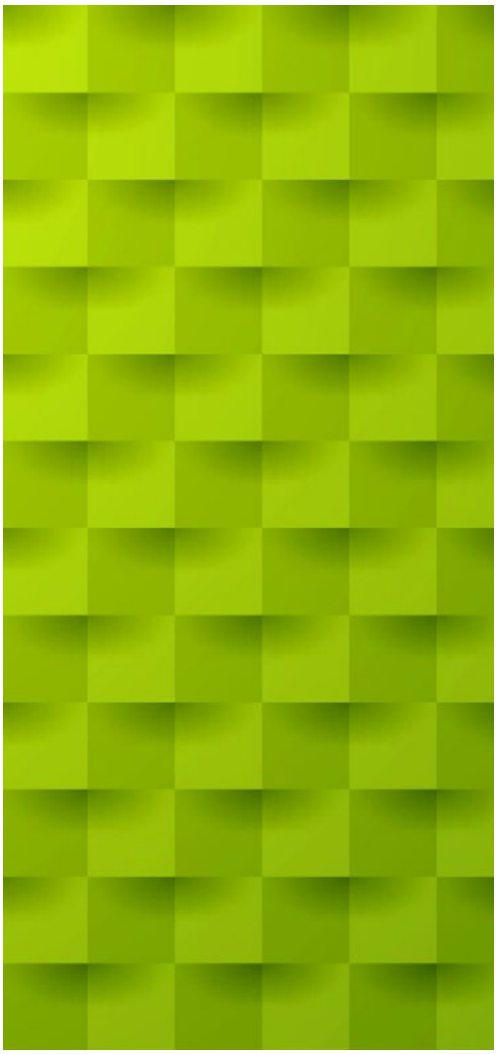


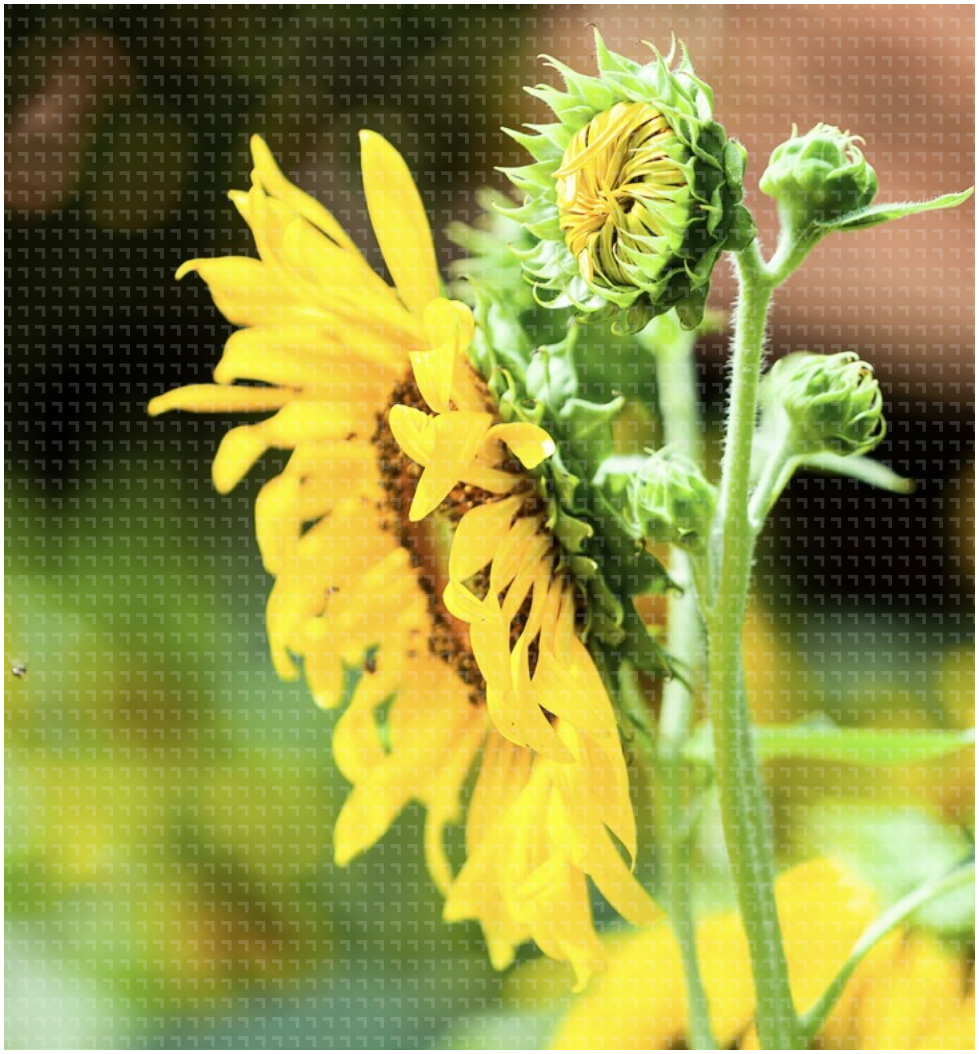
A greener architecture

Regardless of how straightforward or complex the software, how strict or simple the requirements, it is essential to initially develop a vision of the expected outcome. This can range from a simple conceptual sentence to a detailed architectural document. Whatever approach is taken, it will form the basis of all subsequent decisions.

As a result, challenging the plan will probably have major implications in terms of costs and timings (which also implies energy may be wasted).

We will now explore some of the overarching principles that will help define a more comprehensive, sustainable vision for sustainable development.





Shut down when idle r

The basic principles of saving energy at home can (and should) also be applied to software design. In the same way you should turn off the lights when no one is in a room, you should shut down software when nobody is using it. In keeping with this analogy, note that we simply turn off the lights rather than pull plugs out or deactivate the power for the entire house. Accordingly, applications need to be designed based on modular principles so they can be shut down separately.

This approach is central to microservices and serverless architectures – not only when it comes to total shutdowns, but also with respect to scalability or starting and stopping replica modules around the world due to demand fluctuations. Decisions based on this approach will be reflected in the final implementation and deployment (also covered later in this document).

Striking the right balance can be a fine art, especially if you have no comparable experience to draw upon. If that's the case and you're dealing with a new product, go live anyway and track actual usage patterns to see how systems behave and identify potential optimisations. Regardless of how much you already know about demand levels, your code should always be in a position to separate individual sections and transform them into new and autonomous modules. This is already highlighted as best practice when writing code for such designs, but it is also useful to envisage how you want to split individual sections separately, so you don't unwittingly keep components coupled.

Avoid impulsive consumption

Delving deeper and analysing internal services, another way to enhance efficiency may be to prioritise resilience by gearing certain sections of the software towards potential unavailability.

One common approach is to add procedural asynchrony; deliberately 'misusing' systems and collating jobs for clustering and processing together and in sequence. If tasks are not checked to see if they require real-time processing, all processing will automatically be carried out in the default real-time mode. Challenging whether real-time processing is necessary may conclude that end-of-day processing could be the most valid and efficient approach to adopt. Even once-per-hour processing could make a difference, especially if requirements allow for volumes to be processed in consolidated batches at a later point in time.

One could also apply this concept to deliverables in order to improve overall efficiency. Some operations that are completed at runtime could also happen during build time. One example is code introspection (allowing certain sections of code to inspect others and / or generate new code at runtime), ensuring there is no chance of performing code generation during the build when the final, executable file is being generated. Another example could be the landing page of a web application; where the inner sections may be based on truly dynamic content, but also the initial landing content may change every day. In this situation we may consider using static site generation techniques, re-building the page once (per release, per week, per day or even per hour, depending on your needs) and transforming any dynamic content into static content that is much easier to optimise in terms of energy consumption. These examples (and many other scenarios) could also be dealt with by utilising caching along the lines of 'lazy build' procedures. There is nothing wrong with solving problems in this way, as long as we continue to challenge the 'just-in-time' requirements.

Focus your time and energy investment

When integrating green code criteria into an architectural design, the lifecycle of the software must always be considered; its creation, use, maintenance and disposal. To see the big picture, we must always to consider the ultimate target audience for the software. The first consideration will be to examine factors relating to humans or machines; will it be used by human users or other systems? How many users do we expect - dozens, or maybe thousands? These lines of enquiry must continue to ensure that the expected usage frequencies and the duration of average software interactions is properly understood.

This analysis should provide a good understanding of which elements within the architecture will require the most energy. For example, with back-office software hosted on a shared server, it is not unusual for any development to generate a major energy footprint if the software is actually only used for a handful of minutes per week (if at all). This contrasts, for example, with a simple timetable application used by a university, which may be required to generate thousands of images every hour. Saving even 0.1 of a second to create such an image could save a significant volume of energy every year.

A good example of how even a small time saving can have a huge impact could be optimising the startup time for a virtual banking app used by 500,000 customers. By replacing loading screen images and reducing their resolutions, opening times may be reduced – even by one millisecond. Assuming the average user opens the app at least once a day, this could save over 50 hours (or more than 2 days!) of operating time on mobile devices per year.

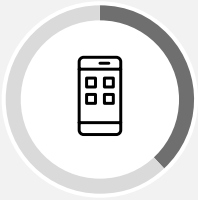
The numbers we are discussing here should already be central to the process of understanding non-functional requirements and estimating initial efforts and time investment in any project. From a GreenCoding perspective, it makes sense to come up with actual comparisons of the time investment for each phase of the development. This can be as simple or as complex as required, but for the sake of clarity, we will examine a relatively simple use case. If we examine the energy footprint of a team manager, a developer and a tester using a laptop, we can them 5 each 'units'. We can also provide the server with 20 units, whilst each mobile device user will account for 1 unit. This initial approach may be oversimplified, but even such a simple model will give you a good general idea. If you want to make it more granular, you can easily add as many layers as you want.

DELIVERY

Total (human) working hours required to make the application available

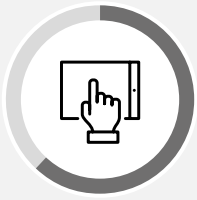
PROJECTED MAINTENANCE

Total working hours per year



RUNTIME

Server uptime(s) per year and over the expected lifetime



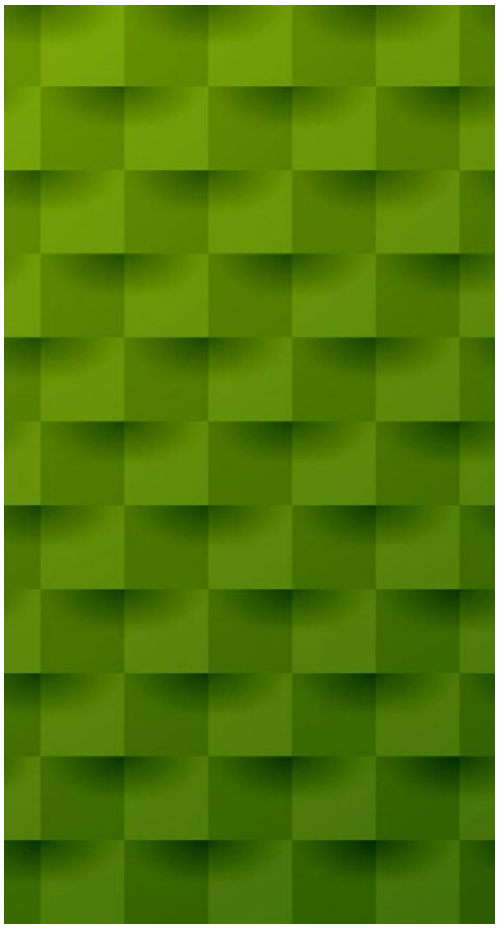
USAGE TIME

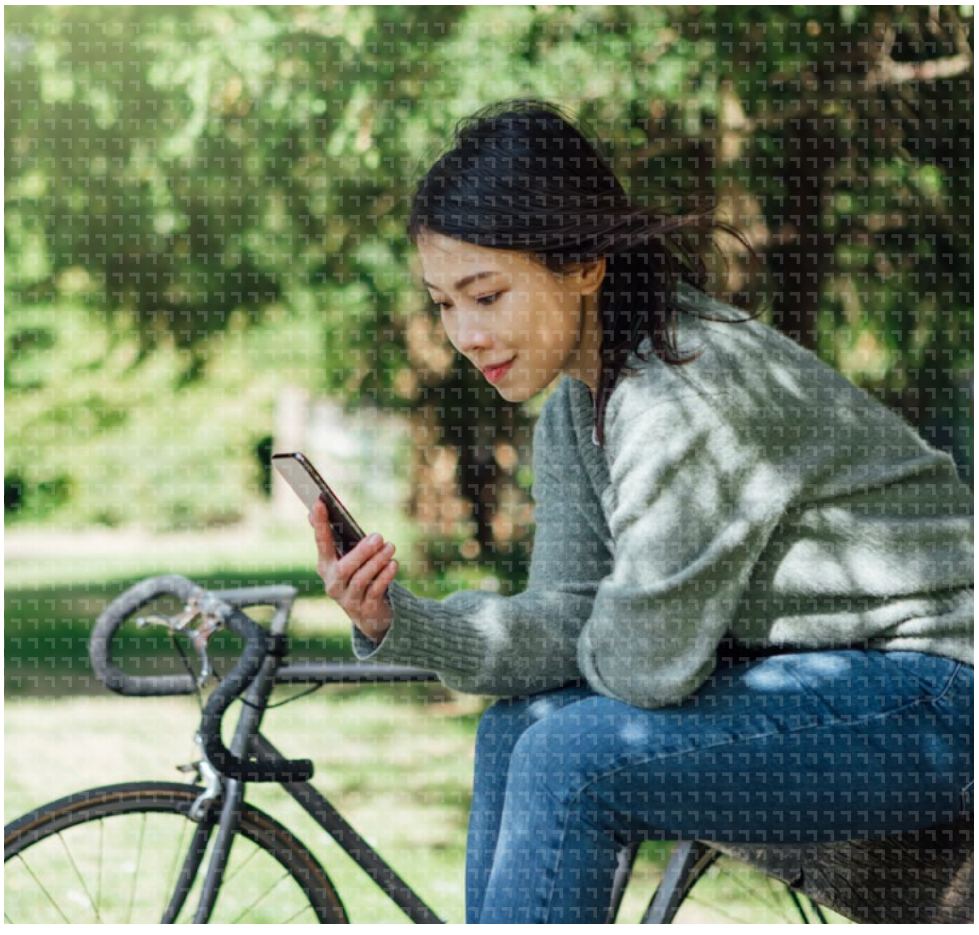
Estimate of the time taken to complete a single user interaction (expressed as a standard use case / customer journey, which is then multiplied by the anticipated number of user interactions per year over the expected lifetime)

WAIT TIME

Estimate of the time spent on the 'Loading'... screen for a single user interaction (expressed as a standard use case / customer journey, which is then multiplied by the anticipated number of interactions per year over the expected lifetime)

Having a visualisation of the time spent on every stage of the software life cycle – even if giving only a general idea – will help the whole software delivery chain gain a better understanding of where to focus the efforts to reduce the energy footprint.





Adopting greener logic

Once the architectural foundations are set, it is time to then make things tangible. It will often appear that everyday development decisions have little impact on overall performance, but actually that is not the case and one should avoid falling into this particular trap. Every decision matters. It is rare to be lucky enough to cut loading time by 20% just by carrying out two hours of recoding, however, sometimes this may be possible. Nonetheless, these efforts are required if the goal of implementing sustainability into software design is to be achieved. Likewise, even if the impact of a certain decision may seem negligible in isolation, the ramifications of an individual decision should never be ignored as these decisions multiply. Ten 'virtually negligible' impacts can combine to make a noticeable difference. Performance always matters, regardless of scale.

The aim of this section of the paper is to provide the reader with some ideas on how to pinpoint the potential performance enhancements from GreenCoding. Some of these may apply to a live project or on a future planned development. Some may deliver huge benefit with certain types of software. With others, the time and effort invested will not be worth the savings that are realised. The important thing is that everyone be aware of these issues and that they are a focal point for future improvements.

Zero waste code

From a technical and delivery perspective, coding is fundamentally about solving problems. Originally, firms developed bespoke customised solutions for each problem and although it did not take long for businesses to start offering publicly available software, the actual revolution came with open-source software and the free distribution of licences. Pieces of code were made publicly available, ready to solve large or small problems, code that could be further combined and extended.

As a result, delivery times and resource investments have reduced significantly over time, which is of course a positive development. Nevertheless, as in the physical world of environmentalism, just because something is free, it does not mean it should be used irresponsibly. As much as 90% of modern software contains open-source code developed by third parties and whilst this is not necessarily bad, potentially the problem to be solved may be an exact fit with a pre-existing library design. In addition, as more external resources become available, there is an increasing chance that redundant sections of code will be introduced (or may already have been).

This is particularly important with website applications that require software to be downloaded and installed every time a user visits. Although code may be 'free to consume' by developers and potentially has negligible impact on build times, if code is used but not actually needed by users to gain the benefits they expected, it may waste network time and CPU parsing time. It should be remembered that browsers will parse any information they are given so they can execute it when needed, therefore it is the responsibility of web developers to ensure workloads are efficient.

For mobile or desktop applications, network time is less relevant if applications are only downloaded once by each user. The initialisation time will, however, still be affected by the bundle size.

Software that is always running on dedicated servers is much less dependent on bundle size, because it is only installed once per quarter, for example. The volume of data that is loaded to start software has less impact on overall processing, but if the requirement is to move to cloud infrastructures or use microservice architectures, it is important to consider how much effort is required to move the application from one server to another. Clouds and clusters do make energy consumption more transparent, but, once again, this does not mean their carbon footprint should be underestimated.

Startup times are still relevant and this can affect the entire strategy; it may be deemed appropriate to shut down a non-critical API because its startup time is one second and it does not exceed the assumed delay to the operator of roughly one second. If the delay were much longer, for example five seconds, that might not be acceptable. In this case the API should never be shut down or should be scaled more appropriately.

Again, issues can be approached from a **preventative** or a **corrective** angle.

Bundle size issues can be prevented by allocating a size budget (or a performance budget). This involves defining how big applications should be and introducing automatic checks to warn developers the instant a budget is depleted. Budgets can also be used to avoid large increments being made for individual features. This allows developers to focus on current required changes rather than having to scan entire projects to identify potential issues. This approach is also useful if the requirement is to quickly discount various options if the first proof of concept has a significant impact on overall size.

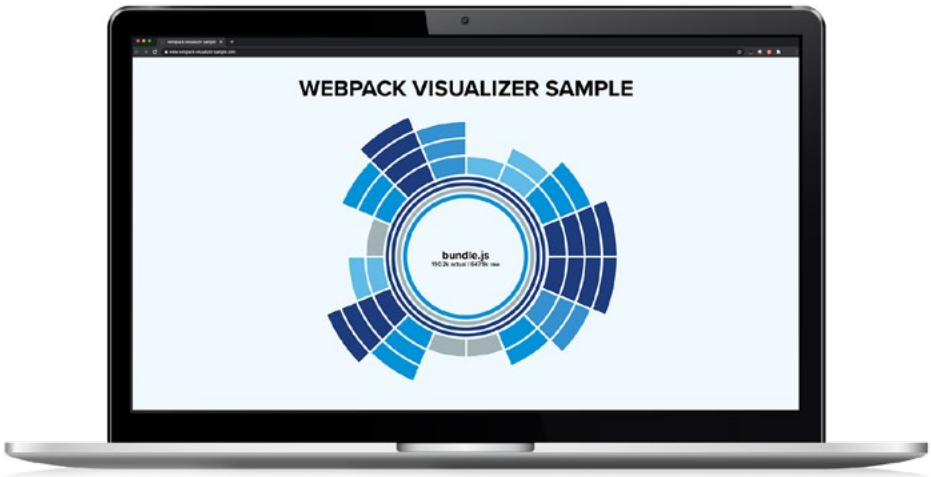
If a project is already live, there always exist ways to introduce corrective measures to reduce bundle size.

The first step is to focus on any code that will never be executed. As mentioned previously, attention should be given to lines of code not written by the team, but referred to as a dependency in order for the code to run. Doing this manually can be hugely time-consuming and extremely risky. A good way to locate and remove 'dead' code safely is therefore to use 'tree-shaking' engines.

The impact of bundle size is not the same in all areas; for example, it is more common for such engines to be applied to computer languages used for web development (such as Javascript, where all major bundlers include this feature by default). It is also possible to find such tools used for typical backend languages (e.g. ProGuard for Java and Kotlin). Nonetheless, this is a very sensitive technique that requires the right approach with respect to the library (providing a fine-grained modularisation), the developer (who will need to make accurate references to library modules) and the tree-shaking engine itself (combining all of the information based on a smart approach).

Tree-shaking engines are very sensitive, therefore everything is dependent on the developer and their attention to detail. Introducing a third-party dependency to a project has to be carefully thought through. It is similar to inviting someone as a guest into your house. At the very least, developing sustainable software requires a clear idea of the impact each 'guest' may have on the final outcome. Of course the web developers will have better insights into the delivered code e.g. they will have more advanced tools such as bundle visualisers, which provide a visual map (boxes or pie charts) showing the relative size of each section of the code (including code borrowed from open sources). Other coding languages may not provide such focus, requiring more manual coding. This also makes it possible to modify third-party libraries instead of only adopting library content without adjustments. Ultimately, 'old-school' techniques such as accessing a library folder and charting file sizes can also work well.

Image:
Bundle visualizer example



Having grasped the impacts each dependency has on your software, one can then prioritise where to make efficiency improvements. Sometimes the benefit-to-impact ratio is poor, so the likelihood of success in replacing the 'unbalanced' library with an alternative (if it cannot be adapted) will need to be weighed up. Of course the library itself may be absolutely fine the way it is designed, but it may just be a poor fit with the needs of the system (for example, perhaps using a huge charting library to draw a single bar chart that could be created manually). At this point, it is worth mentioning the sustainable use of open source software, because this is currently fuelling a stronger focus amongst open source providers on 'footprints', especially with respect to tree-shaking capabilities.
In the same way that supermarkets started introducing environmentally friendly products (and consumer awareness shaped the whole delivery chain), GreenCoding also has the potential to create a paradigm shift that could shape the whole scene of open source solutions.

Low-footprint resources
r

Although developers should initially focus on code, which can be energy-intensive in processing terms, there is always potential to make optimisations by looking at other resources required by your software.[9]

For example, how information is actually organised can have an impact on software. Naturally, this depends on how such resources are used. It may not take much effort to parse an inefficiently structured file once a week, but if it has to be transmitted over a network hundreds of times every hour, this is a clearly far from ideal.

Again, awareness is the hardest part. The obvious option is to think about using different file formats. Maybe an Excel spreadsheet can be replaced by a simple CSV file. Or an XML file can be stored as a YAML file. Of all application resources, there is one that probably stands out most when it comes to overall impact: images.

One of the most overlooked issues with images is how they're packaged. The first decision that will need to be considered is whether to use raster images (as with bitmaps) or vectorial images (based on simple lines and shapes). A good rule of thumb is to use raster images for photos or detailed drawings and to use vector images for logos, symbols and charts. Raster images should be properly sized. Using a detailed 10 MB image for a thumbnail reference is a huge drain on resources. Note that image processing has moved forward in leaps and bounds in recent years and certain new generation image formats (such as webp) have been designed specifically for network transmission.

Vector images were designed with scalability in mind, so although it sounds good, there's no such thing as a default size. Regarding format, SVG is the de facto standard for vector images and it can hardly be bettered. That said, we would always advise developers to optimise networks and processing by clustering vectorial images into single files (using sprites).

Staying on the topic of visual content, but thinking about things from a different angle, visual impressions can have an impact on energy consumption. The emerging concept of dark design is not only changing design preferences but is also unveiling new ways to reduce the amount of energy used by displays. Combined with OLED display technology, which is mainly used on smartphones, dark mode can reduce battery use by up to 23.5%.[10] Accordingly, offering dark mode should be a high priority, especially if you're developing a web or mobile application. Of course this also has implications for marketing and brand design. Giving users the option of alternative (darker) colour schemes can strengthen branding by adding features that are consciously activated by the user. Naturally, some users will be more inclined to respond to such options than others, but you can promote adoption by suggesting switching to dark mode under certain lighting conditions.

[9] "The cost of JavaScript in 2019": https://v8.dev/blog/cost-of-javascript-2019, June 2019

[10] "What is the impact of Dark Mode on battery drain": http://mobileenerlytics.com/dark-mode/, August 2019

Proximity feeds

Whilst software is designed to deliver benefits for the user, dealing properly with information expiry dates can have a major impact on required resources. Any information provided to the user must be classified in terms of its required duration; information can be valid for seconds, sometimes it is needed for days, weeks or even for an unlimited period of time. Combining this aspect with how frequently information is requested gives clarity for caching needs and potential efficiency gains.

Caching (storing relevant data temporarily in an intermediate layer) can be done at several points in the information flow:

The closer the cache is to the user, the better. Sometimes it is useful to apply caching to multiple layers.

It is worth mentioning here the impact that social media and distributed storage have had on how up to date users expect information to be. Efficiency gains may have an impact on the user experience, so for example, it may be worth revisiting the approach to images being delivered to users. What do users look for? Do they need a full view or they are just looking for some results? What will be the impact of holding back some images for a minute or two? Or waiting an hour? Or even a day?

It is surprising how even simple compromises can significantly improve efficiencies and how often it is entirely acceptable to make such updates.
In terms of practical implementation, however, it is advisable to have a data or information architect analyse all data feeds during the design phase. That said, modern applications are based on layered or distributed architectures, which should provide detailed ongoing usage metrics. Analysing metrics and identifying potential feed optimisations in the live environment (by locating the most frequent requests and origins) would be the best way to make improvements, based on actual user behaviour.

This is a particularly important with web applications, since such systems transmit their own interface software on each and every user visit. There is also a wider scope of potential optimisation in this area, since it is easy to identify ‘almost permanent’ content (e.g. a logo). As a result, a couple of concepts emerge in addition to regular HTTP caching:

01.
Progressive web applications (PWAs): modern browsers can transform web pages compatible with PWA standards into applications. This technique provides more elaborate logic capabilities in terms of content expiration handling, as well as offline support.[11,12]

02.
Content delivery networks (CDNs) in edge locations: a CDN is a highly distributed platform of servers that helps minimise delays in loading web page content by reducing the physical distance between the server and the user. CDN providers have created smart and adaptable solutions to enable easy enrolling of web applications and in some cases, they can also act as an additional caching layer for data between services and the visual interface.

[11] “Impact of Progressive Web Apps on Web App Development”: https://www.researchgate.net/publication/330834334_Impact_of_Progressive_Web_Apps_on_Web_App_Development, September 2018

[12] “Progressive Web Apps for the Unified Development of Mobile Applications”: https://link.springer.com/chapter/10.1007/978-3-319-93527-0_4, June 2018

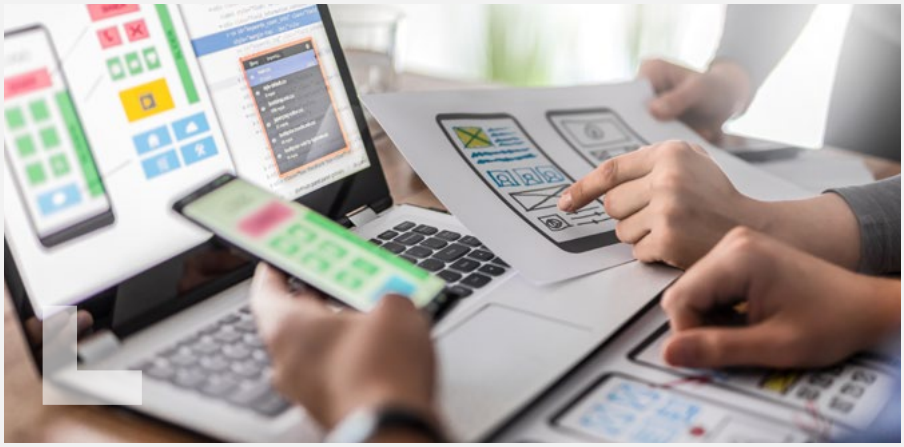
Greener methods

Rapid feedback, better decisions

One of the major challenges to succeed with GreenCoding is awareness. All the techniques previously described are natural responses to increase efficiency once stakeholders become aware of the amount of energy that is being wasted. As a result, the sooner a team becomes aware of the energy impact of their decisions, the easier it will be to improve decision-making. This points directly to feedback methods, which need to be organised into rapid cycles.

On an overarching level, agile and lean methods provide better opportunities to adapt software to the principles of offering benefit-driven screens, as described in the previous section.[13] On a deeper level, applying continuous integration and continuous delivery allows you to visualise the final impact of each development decision.[14]

Once a team has rapid feedback cycles in place, it will need to define any metrics to be used, based on any assumptions regarding Green Code. Of course there are plenty of complex metrics that could be monitored, but the best return on investment is most likely to with loading times. These are easy to measure (and watch with the naked eye) and directly correlate with energy consumption. Teams should keep records of initial loading times and for main interactions. Obtaining such metrics is usually standard practice with projects, but only in the final stages in order to ask, 'is this functionally acceptable?'.



This mindset needs to change to improve efficiency – even if loading times may be acceptable from a usability perspective, if an application were scrutinised according to GreenCoding principles, slow loading times would not be considered sustainable.

Tracking loading and interaction times from the outset is a highly effective way to focus on sustainability.

In practical terms, the development team should start with the basics (i.e. 'Hello world' loading times) then tag each new feature and its impact. Obviously, initial functions such as accessing storage will have a significant impact. It is essential to ensure there is a clear picture of the before and the after situation so, whatever the impact may be, that there is enough information to properly evaluate if the value provided by the change delivered a worthwhile benefit. As long as significant increments are challenged, sustainability objectives will be met. On rare occasions, there may be obvious resource-intensive routines that play a defining role in loading or interaction times that unwittingly conceal underlying inefficiencies. To reveal such inefficiencies, scenarios should be created where wasteful processes are replaced by almost instantaneous 'pre-recorded' responses, providing a secondary set of performance metrics that are unaffected by the resource-intensive routines.

Whilst developers are encouraged to apply continuous integration (CI) and continuous delivery (CD) techniques to support rapid feedback cycles, attention should also be given to configuration. Careless configurations of CICD tools may lead to complex integration tests running automatically more often than desired or even required. It is essential to ensure that development teams strike the right balance between gaining automatic feedback and maintaining sufficient resources to process or react to feedback. One way to achieve large saving, both in terms of direct energy consumption (CPU time) and indirect energy consumption (time spent by developers waiting for feedback), is to apply incremental building techniques.[15] This involves only recompiling modified sections of code instead of entire deliverables, or only testing rewritten code and systems affected by recompiled code. Build outputs should also be shared between developers and CICD systems.

[13] "Lean and Agile: differences and similarities": https://twproject.com/blog/lean-agile-differences-similarities/, November 2018

[14] "Continuous integration vs. continuous delivery vs. continuous deployment": https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[15] "Incremental Model in SDLC: Use, Advantage & Disadvantage": https://www.guru99.com/what-is-incremental-model-in-sdlc-advantages-disadvantages.html, November 2020

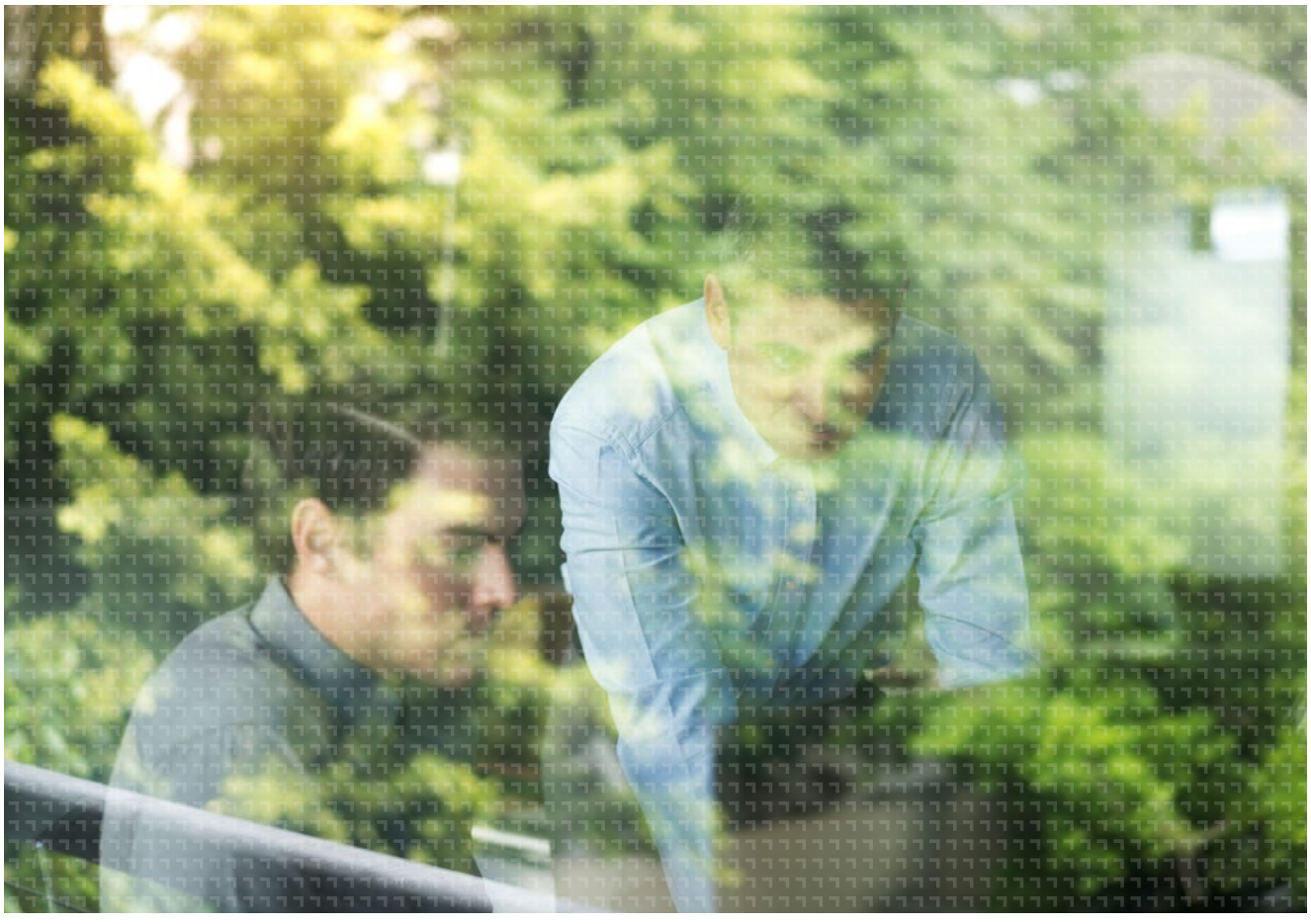
Reusable output

Understanding the opportunity to make software more sustainable does take time and effort, but it is an important starting point. Teams, companies and business units are heterogeneous by nature, and as a result, they will also naturally adopt different approaches to GreenCoding. It is entirely possible that organisations will need to introduce and examine benchmark data for a number of days. In other instances, it may only take a couple of minutes and a few simple sketches to understand what is happening. Either way, in most cased the team will have started in a position of uncertainty, faced with challenges and new requirements, involving time and energy being invested into generating new output. Best practice is to organise and share this in a way that is accessible and legible to other team members, the entire organisation, or even the wider IT community.

Part and parcel of sustainable software development is ensuring that the results of GreenCoding projects are available to others and can be searched by team members, people within the organisation or even the community at large, and this procedure should also remain efficient in itself.

Leaving the community of developers to tackle the same issues and problems again themselves means having to start from scratch (even on a GreenCoding project), which clearly wastes time, effort and therefore energy.

It is important to highlight the positive impacts of making efficiency changes to projects for a number of reasons. Establishing metrics regarding the efficiency contributions made by undertaking specific actions within projects, such benefits can then be extrapolated to other projects with similar conditions. Having such metrics sidesteps the need to quantify benefits a second time by carrying out benchmarking, with new actions being based on assumed net gains during the design process. Ultimately, the objective for any team applying GreenCoding principles should be to minimise the number of required corrective measures, by drawing on a reliable arsenal of preventive measures. As highlighted above, project contexts are heterogeneous by nature, so nothing in this area is merely ‘black or white’; success ultimately depends on the ability to capture and index customised sets of preventive measures that will be relevant to development projects.



A greener platform

Examining GreenCoding from a broader perspective, considerations regarding the right infrastructure for running code are as crucial as the code itself. When it comes to hardware and computing power, utilisation levels are crucial. Why do utilisation levels have such a strong influence on energy efficiency?

Optimal utilisation

The energy consumed by a computer system is not proportional to utilisation levels.

This concept is known as energy proportionality, with higher utilisation levels resulting in lower energy consumption per percentage point of utilised computing power.

Low server utilisation rates are a common problem. They are usually caused by a tendency during planning to overestimate how much software and therefore server capacity will actually be used. For example, developers may anticipate too many users or expect more users to come on board later down the line. Based on their incorrect assumptions, computing power is often extrapolated over several years resulting in significantly oversized systems. Since most systems run multiple applications, it can be impossible to pinpoint energy consumption for a specific application, such as a program running on a shared monolith, alongside several other applications.

One effective way to work out what is happening and to track the energy consumption of a specific application is to use the cloud, not because cloud providers are better at maths and have more accurate numbers, but simply because in an operational sense you wate less resource. With cloud computing, your see straightforward correlations – the higher the invoice, the more energy was consumed.

More money spent on cloud computing usually equals more energy consumed.

Using the cloud can also significantly influence energy consumption. As described earlier, higher hardware utilisation levels are also more energy-efficient. This is why cloud computing offers a useful potential to save energy. Even when your internal / on-premise servers are idle, they are still using energy. Public cloud systems are based on principles of high modularity, making it possible to control utilisation levels more precisely than with non-modular systems, especially if you are running modules that cannot be switched off to stop them consuming energy when they are not in use.

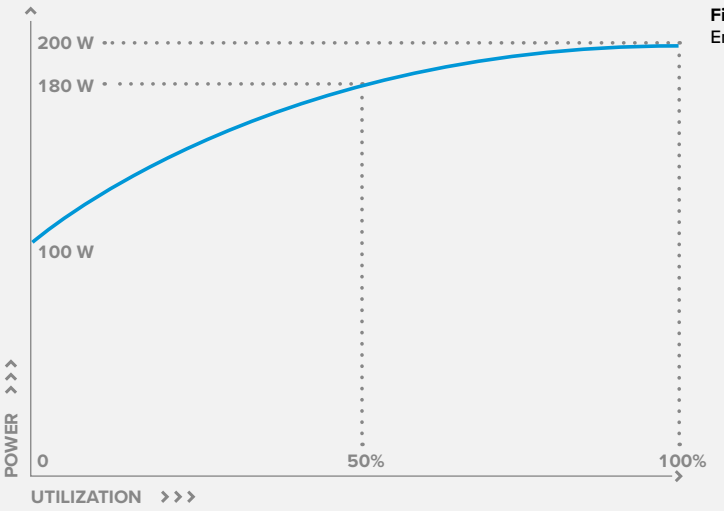


Figure 3:
Energy proportionality

According to a paper published by the Natural Resources Defense Council (NRDC), the cloud server utilisation levels of large providers such as AWS, Google Cloud and Microsoft Azure stand at around 65%.[16] This compares to on-premise data centres, which typically run at utilisation levels of between 12% and 18%. Keeping in mind the previously mentioned concept of energy proportionality, this equates to significant disadvantages in terms of energy efficiency.

Another important aspect when it comes to improving energy efficiency is the design of technical infrastructure, for example, cooling technology. Large investments in system efficiency will transform into important cost reductions for providers, offering plenty of motivation to merge environmental gains with economic benefits. Large-scale infrastructure improvements can help reduce power consumption by up to 29% versus typical on-premise data-centres.[17] Google Cloud has taken efficiency one step further and now uses machine learning to reduce energy required for cooling purposes by up to 40%.[18]

The final part of the equation when it comes to minimising the carbon footprint of cloud computing is the energy source, or so-called power mix. One goal that all public cloud providers have in common is to be solely reliable on renewable energy for powering cloud infrastructure. Some are closer to achieving this goal than others.

High utilisation levels, efficient infrastructure design and a clean power mix are thus key drivers when it comes to minimising the environmental impact of cloud computing. These are areas where the large cloud providers have clear advantages. Even if the technology they offer is energy-intensive, they are in a position to systematically improve infrastructure and make cloud computing more efficient.

[16] NRDC, "Data Center Efficiency Assessment": https://www.nrdc.org/sites/default/files/data-center-efficiency-assessment-IP.pdf, August 2014

[17] AWS News Blog, "Cloud Computing, Server Utilization, & the Environment": https://aws.amazon.com/blogs/aws/cloud-computing-server-utilization-the-environment/, June 2015

[18] Google Blog, "DeepMind AI reduces energy used for cooling Google data centers by 40%": https://blog.google/outreach-initiatives/environment/deepmind-ai-reduces-energy-used-for/, July 2016

Precise configuration

Any product used these days is likely to run without any issues under its default configuration. Despite this, running software on a platform with default configuration settings (regardless of whether it is based on a server, a container or using serverless methods) is like wearing shoes without tying up the laces; they may feel okay, but are extremely unlikely to fit properly. Similarly, ill-fitting platform configurations are another way of wasting energy. Of course it depends on the platform and how much flexibility there is to fine-tune configurations, but it is desirable at least to understand the options and the implications of using the default settings.

For example, by considering the configuration options it may be possible to uncover inefficient and suboptimal network communications; perhaps HTTP2 and / or gzip compression settings were never enabled? Similarly, a Java virtual machine may be struggling to deal with garbage collection because of insufficient memory allocation. Perhaps there is so much information moving around within headers across internal or even external networks, that transmission levels are almost double what they need to be?

Unfortunately, it is very rare for developers to look into these topics in depth until a performance issue crops up, and when they do, they may only be in a position to make 'tweaks' to alleviate the current problem. Having to quickly resolve the issue is then a missed opportunity to achieve something better. Sustainable software delivery should keep a close eye on platform capabilities, not just at release, but also during the entire lifetime of an application. It is often said that "if it's not broken, don't fix it", but for the greater good, it does sometimes make sense to challenge this first principle of computing.

Holistic metrics

As outlined earlier, it is crucial to focus efforts on the right areas and strike the correct balance between invested time and efforts on one hand, and performance / usability enhancements on the other. The challenge is that often not all the required and accurate information is available, therefore decisions have to be made on averages and projections. The first step should therefore always be to use what data points are available, but in the longer better tools and metrics will be needed in order to refine the strategy.

Regardless of whether servers are on-premise or part of a cloud arrangement, they are already a focus area for energy reduction efforts. This is especially relevant for cloud providers that own massive server infrastructures and therefore can leverage bigger savings from any improvements they achieve. This enables them to dedicate significant efforts (in time and money) to investigate new ways of reducing energy consumption. In parallel, they also track the ecological impact of energy they consume in order to reduce their overall CO_2 footprint.

One aspect that often gets overlooked when assessing system efficiency is that of 'hidden' infrastructure, namely personal devices. For many, personal devices are only important when it comes to customer satisfaction. However, from a holistic standpoint, they are very important for software developers because they are prolific, connected and add to the overall energy footprint.

By their very nature, the laptops, tablets and smart devices used will be extremely diverse, as are the users themselves and their individual behaviours, which combine to create a very complex model. Of course a focus on the user experience is key, and specifically we can investigate how 'green' is the energy that feeds the end-user devices. Energy costs in different countries can be examined to understand the impact of wasted energy relative to GDP, user income or other metrics. We must also overlay that technology landscape for users in the developed world will very different from those in underdeveloped nations; saying that, even in developed countries network bandwidths in remote and rural areas can very unreliable and unstable.

If the development community can align behind a common understanding and support the primary objectives of Green Coding, it is certain to come up with innovative solutions and inventive ways to tackle the new challenges we face on a global basis.

The virtues of GreenCoding

Once a GreenCoding perspective is established within an organisation, we have seen that the principles of GreenCoding can align very well with the iterative cycles and data-driven methodology of Agile development. This development approach provides the perfect opportunity for the principles to be assimilated into every part of the CICD, process thereby transforming how software is delivered.

However, this approach to software design is in its infancy as within Green Coding's current methodology, there is one particular problem that needs addressing: it can prove to be time-intensive which would then mean projects may take longer than expected and therefore project costs might increase.

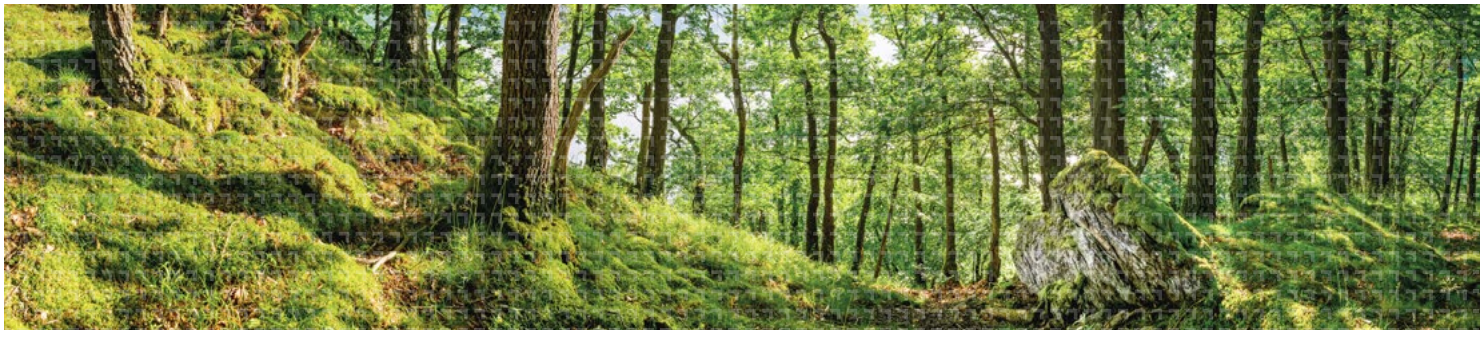
As such, this paper is specifically designed to challenge current thinking and attitudes towards software development, as well as provide examples of how the environmental impact of the entire software life-cycle can be reduced. It must be stressed that the climate emergency can only be solved through widespread collaboration. Taking a collaborative and problem-solving approach and innovating current processes within the tech and IT industry, it us up to individual organisations to pave the way forward. As an innovator with the software development industry, GFT has outlined a blueprint, whereby as in other sectors and industries, Green Coding has the potential to do more than simply reducing emissions.

Specifically, the GreenCoding approach for software development can be summarised as follows:

- Improving software by making it more energy efficient also has the potential to make it easier to use and faster.
- Enhancing the user experience, and allowing software to be offered to an even wider target audience
- Operational software costs become more efficient, resulting in important cost savings for companies.

The concept of GreenCoding is still in its infancy and ideally something that can be put into practice over time; a philosophy that can be gradually implemented, provided that time expended on individual projects are not overly extended. It is evident that GreenCoding has great potential in starting a global movement amongst developers, and can be considered as a bold new 'green frontier' for software developers.

Whilst the efficiencies of cloud computing has been a major catalyst of change, in the future, GreenCoding is how all software can be made, especially in a global context where we are all striving towards a sustainable and connected planet.



Our authors

Gonzalo Ruiz De Villa Suárez
CTO at GFT

Gonzalo is GFT's Chief Technology Officer and is responsible for the company's technology and innovation strategy, by incubating emerging technology initiatives globally, leading R&D projects in the GFT lab and implementing the latest innovations with clients and the partner ecosystem. Gonzalo is responsible for the GreenCoding initiative at GFT building incremental, transparent and trusted insights on how the IT industry can make a difference for our climate.

Benedict Bax
Executive Assistant to the CEO at GFT

Benedict is the Executive Assistant to the CEO at GFT Technologies offering comprehensive support to the global management team covering all areas of the business. Their experience ranges from technological development to business-strategy thus demonstrating an interdisciplinary approach to the environment and technological sustainability.

Alejandro Reyes Ferreres
Software Architect at GFT

Alejandro is a Software Architect at GFT Technologies and has a deep technological understanding of API's and Frontend development.

With his professional background he has continuously developed the GreenCoding initiative at GFT offering original ideas in its application as well as proofing GreenCoding's key concepts.

About GFT

GFT is driving the digital transformation of the world's leading companies in the financial and insurance sectors, as well as in the manufacturing industry.

As an IT services and software engineering provider, GFT offers strong consulting and development skills across all aspects of pioneering technologies, such as cloud engineering, artificial intelligence, mainframe modernisation and the Internet of Things for Industry 4.0.

With its in-depth technological expertise, profound market know-how and strong partnerships, GFT implements scalable IT solutions to increase productivity. This provides clients with faster access to new IT applications and innovative business models, while also reducing risk.

Founded in 1987 and located in more than 15 countries to ensure close proximity to its clients, GFT employs 6,000 people. GFT provides them with career opportunities in all areas of software engineering and innovation.

The GFT Technologies SE share is listed in the Prime Standard segment of the Frankfurt Stock Exchange (ticker: GFT-XE).

blog.gft.com
twitter.com/gft_en
linkedin.com/company/gft-group
facebook.com/GFTGroup
› gft.com