The Ethical Imperative of Efficient Computing

- or why making efficient software is an ethical obligation.

Introduction

Modern society runs on software. Every email sent, every document edited, and every click of the **Start Menu** represents not just a moment of productivity, but also a consumption of resources – energy, hardware capacity, and human time. As hardware engineers have pushed the limits of efficiency (from low-power chips to energy-saving devices), software has moved in the opposite direction, becoming more **resource-intensive** without proportional gains in user benefit[1][2]. This imbalance raises an ethical question: *Do we have a responsibility to make computing more efficient?* The answer is yes. Ensuring our software is efficient is not just a technical goal, but an ethical imperative, because inefficient computing at scale harms the environment, wastes energy, and erodes human productivity and well-being.

Efficient computing here refers primarily to software efficiency – delivering the features and performance without consuming more resources than needed. Unlike hardware (which has strong market and physical incentives to improve energy efficiency), software efficiency has lagged behind. Why is this an ethical concern? There are two major reasons: **environmental impact** and **human impact**. Environmentally, bloated and inefficient software leads to higher electricity usage and faster hardware obsolescence, contributing to carbon emissions and e-waste. Socially, slow and inefficient software wastes countless years of human life in aggregate, causing frustration and lost productivity. This paper explores the historical context of software efficiency, examines the troubling trend of modern software bloat, quantifies the environmental and human costs of inefficiency, and argues that closing the efficiency gap is a matter of ethical responsibility for software creators and policymakers.

Historical Context: Doing More with Less

In the early decades of computing, software **had** to do more with less. Hardware was limited – processing power was measured in megahertz and memory in megabytes, sometimes kilobytes – programmers were simply forced to optimize ruthlessly to achieve anything at all. This era left us with a valuable lesson: many of the core tasks people need (office productivity, email, basic data processing) are

achievable with far fewer resources than modern software demands. For example, in the late 1990s, a standard office computer running Windows 98 or Windows 2000 with Microsoft Office, could handle email, word processing, and spreadsheets – essentially **the same tasks that dominate office work today** – on hardware thousands of times less powerful than today's machines. If not for modern software incompatibilities and online requirements, those systems could still enable a large portion of today's office work[3][4]. This historical perspective reveals that *progress* in software has not always been about necessity; often it has been about taking advantage of abundant resources.

Computer scientists have long observed a troubling pattern: as hardware gets faster, software often gets **slower** at nearly the same rate, resulting in no net gain to the end user. This observation is famously codified in **Wirth's Law**, which states: "software is getting slower more rapidly than hardware is becoming faster"[5]. Niklaus Wirth noted in 1995 that software tends to grow in complexity and resource demands, often outpacing the improvements in hardware speed. In his essay *A Plea for Lean Software*, Wirth argued that many added software features are **"cute but not essential"**, and that people often mistake unnecessary complexity for sophistication – a mistake that carries a hidden cost in performance and efficiency[6]. An old quip captures a similar idea: *"What Intel giveth, Microsoft taketh away."* This refers to how advances by hardware companies like Intel (faster CPUs, more memory) are quickly consumed by heavier Windows operating systems and applications[4]. In other words, each new generation of hardware is immediately offset by more demanding software, leaving users with **the same or even worse perceived performance** despite exponentially more computing power[7].

The historical "last good" example often cited by IT veterans is **Windows 2000** – an operating system that, eventually, struck a balance of performance and simplicity. Since then, many feel that user-facing performance has stagnated or regressed in the Windows line, even as under-the-hood capabilities and hardware support expanded[8]. Each successive version introduced new features and graphical enhancements, but also more background processes, telemetry, and complexity. The result is that tasks like opening a Start Menu or launching a text editor are measurably *slower* on a brand new computer today than they did decades ago on far weaker machines[9][10]. This isn't to say we should literally go back to 1990s software – security and functionality have improved – but the *degree* of inefficiency growth is wildly disproportionate to the tangible benefits in most everyday computing tasks. The historical record makes one thing clear: **software bloat is not an inevitability of progress, but a choice**. We have achieved great things

under far stricter technical constraints; therefore, the rampant inefficiency of modern software cannot be justified as the cost of innovation.

The Rise of Software Bloat and Its Causes

If hardware today is incredibly powerful and energy-efficient, why is software often sluggish and inefficient? The answer lies in the phenomenon of **software bloat** – the tendency of software to become more resource-intensive (using more CPU, memory, disk space, and power) over time, without commensurate improvements in functionality or user experience. Wikipedia succinctly defines *software bloat* as when successive versions of a program become *"perceptibly slower, use more memory, disk space or processing power, or have higher hardware requirements than the previous version, while making only dubious user-perceptible improvements."*[11] In practice, this means that an update or new application version might consume double the RAM and take longer to load, yet from the user's perspective **it doesn't enable much more than the older version did**. This trend has been observed across operating systems, office suites, web browsers, and more.

Several factors drive software bloat in the modern era:

- **Feature Creep and Complexity:** Developers often add features upon features in each release, some essential, many not. The accumulation of "nice-to-have" features can somewhat increase code complexity and size, even though most users will only use a small fraction of the functionality. (It's telling that in Microsoft Word, a **handful of commands account for the majority of usage**, while dozens of rarely-used features still incur a performance and memory cost for everyone[12][6].) Wirth noted that *customers' ignorance of which features are essential vs. optional* allows unnecessary complexity to flourish[6]. In essence, software companies often compete on **feature lists** rather than optimization, under the assumption that more features (tangible or not) add value – even if it slows things down.

- **Abstraction and Heavy Frameworks:** Modern software development emphasizes high-level programming languages and frameworks for the sake of developer productivity. While high-level tools speed up development, they often introduce significant overhead at runtime[13]. For example, many desktop applications today are built using **Electron (Chromium-based) or web frameworks**, which essentially bundle a web browser engine with each app. This approach simplifies cross-platform development (one codebase for web, Windows, Mac, etc.), but at tremendous computational cost: an Electron-based app often uses

hundreds of megabytes of memory and high CPU just to display what might looks like a simple interface[14]. A user on a tech forum pointed out that **Slack, Microsoft Teams, and Visual Studio Code (all Electron apps) together consume nearly 1 GB of RAM when idle**, whereas traditional native apps like Notepad++ or Emacs use under 30 MB for similar functionality[14]. In other words, the convenience of these frameworks means **duplicating heavy browser engines across applications**, wasting memory and processing power for little user gain (especially when the apps are essentially showing text, lists, or simple UIs).

- **Lack of Optimization Incentives:** The rapid growth of hardware performance has perversely reduced the pressure on programmers to optimize. When a new laptop comes with 16 GB of RAM and a multi-core CPU, a developer may not notice (or may ignore) that their code is **inefficient**, because it still runs "fast enough" on their high-end machine. This dynamic was described humorously by former Microsoft CTO Nathan Myhrvold in his laws of software: *"Software is a gas; it expands to fit whatever container it is stored in."*[15] and *"Software growth makes Moore's Law possible: people buy new hardware because the software requires it."*[16]. In short, as soon as hardware provides more capacity, software finds ways to use it up – partly because developers take advantage of the breathing room to write more general, less optimized code and include more features. In the past, code had to be lean to even run; today, bloated code still runs, just silently taxing the system resources in the background.

- **Prioritizing Developer Convenience and Time-to-Market:** In commercial software development, delivering features quickly often takes precedence over fine-tuning performance. Using large third-party libraries or higher-level languages can speed up development, at the cost of including a lot of code that one product might not actually need. Studies have shown that including **many third-party libraries** (common in modern apps) can increase energy use and slow performance, even if it accelerates development[17][18]. The end result is a bigger, slower application, but it arrives to market sooner or with more checklist features – a trade-off that many companies are willing to make, assuming that hardware advancements will cover the difference. Unfortunately, this mindset cumulatively leads to industry-wide bloat.

- **"Good Enough" Culture and User Tolerance:** If users continue to accept slower startup times or heavier applications as a fact of life, there is little

market pushback on inefficiency. For instance, when a typical user clicks the Start Menu in the latest Windows and it lags for a second, they may attribute it to "normal" computer slowness, not realizing that even this simple action has become **unnecessarily heavy**. (Recent revelations showed parts of the Windows 11 Start Menu are implemented with web technologies like React, causing CPU spikes of 30–70% on each open[19][20]. Such design choices are convenient for developers but not necessary for showing a menu of programs – something operating systems have achieved instantaneously for decades.) Yet, because today's PCs have CPU cycles to spare, the inefficiency remains largely invisible except as a slight delay or battery drain. Only when multiplied across millions of users do these small costs manifest as something truly concerning – a point we explore in the next sections.

In summary, software bloat is the product of many small decisions that favor **short-term gains (more features, faster development)** over **long-term efficiency**. Individually, a few extra milliseconds of CPU time or a few megabytes of memory for convenience seem harmless. But collectively, these choices have led to a landscape in which **modern software routinely wastes resources** at a colossal scale. The ethical problem is that these wasted resources are not free – they are paid for by higher energy consumption, carbon emissions, hardware manufacturing, and the time and patience of users.

Environmental Impact of Inefficient Software

Digital technology might seem clean compared to industries like transportation or manufacturing – after all, what is there to see except code and electrons? But the **carbon footprint of computing** is very real, and it's growing alarmingly. When software is inefficient, it causes computers and data centers to consume more electricity for the same tasks, which in turn means more fossil fuels burned (unless all energy is renewable) and more carbon dioxide emitted. Consider these sobering statistics:

- The Information and Communication Technology (ICT) sector as a whole – which includes all our devices, software, and networks – is on track to become a significant chunk of global emissions. A peer-reviewed study projected that if trends continue, **ICT's share of global greenhouse gas emissions could rise from about 1.5% in 2007 to over 14% by 2040**[21]. Fourteen percent of global emissions is nearly half of what the entire transportation sector emits today[21]. Much of this ICT footprint comes from electricity usage by data centers, network infrastructure, and charging billions of devices – and all of that usage is driven by software activity. In

other words, every inefficient app or website, when scaled to millions or billions of instances, directly contributes to climate change through increased power consumption.

- Software inefficiency leads to **wasted energy**. One academic analysis put it bluntly: *"Moving to higher-level programming languages increases the energy consumption. Software bloat and the increasing complexity of digital systems further aggravate the problems."*[13] The authors argue that the major concern for energy use in computing today "is not in hardware evolution but in the way software is written," labeling much of today's software as *environmentally "black" (dirty)* in terms of sustainability[2]. The implication is that our devices could be performing the same computations with far less energy if the software were optimized to do so; instead, cycles are wasted on inefficient code, unnecessary background processes, or bloated web pages.

- A vivid example is the common web page or mobile app. The median size of a website has ballooned dramatically in the past decade – **the average desktop web page grew about 336% (from ~468 KB to over 2 MB) in about ten years**, and the average mobile web page grew over 1200% (from ~145 KB to nearly 1.9 MB) in the same period[22]. This bloat in content (images, scripts, ads, auto-playing videos, high-resolution media) means that even reading news or checking email in a web browser today can use several times more data and processing than a decade ago. All that extra data requires more energy to transmit and store. It's no wonder that internet traffic and data center loads have surged – but disturbingly, much of it is not critical communications but **entertainment and overhead**. Cisco reports that streaming video and entertainment account for over 80% of global internet traffic, whereas web/data (which would include productivity and essential services) is only ~12%[23][24]. Meanwhile, **digital advertising – bloated scripts and tracking pixels that piggyback on websites – may account for up to 10% of all internet energy consumption** by some estimates[25]. This means a substantial portion of our ICT energy use is going into pushing pixels for ads and tracking that the user did not explicitly seek out.

- Even ostensibly simple tasks have a carbon footprint. Take the example of an email. It's just text, but the whole system behind it (data centers, network equipment, the device retrieving and displaying it) consumes energy. According to Mike Berners-Lee (author of *How Bad Are Bananas?*), the

average email – a short text sent between laptops – has a footprint of around 0.3 grams of CO_2[26]. A longer email with an attachment might be tens of grams CO_2[27]. That sounds tiny, but consider that **globally we send around 300 billion emails per day**. If even a fraction of those are unnecessary (think of all the "Reply All" emails or spam), that's a significant carbon waste. In fact, Berners-Lee estimated that all the world's emails in 2019 could have emitted **as much as 150 million tons of CO_2**, about 0.3% of global emissions[28]. The infrastructure for our emails – servers and storage – now has a carbon footprint **greater than the entire pre-2019 global aviation (air travel) industry** did[29]. It's staggering: something as routine as email has scaled to such heights that it's outpacing airplanes in emissions. And part of the reason is inefficiency and unnecessary volume (spam alone was over half of email traffic[30], which is pure waste).

- **Inefficient software shortens hardware lifespan**, contributing to e-waste and manufacturing emissions. When a new version of an application or OS runs intolerably slow on an older device due to poor optimization, users are often forced to replace otherwise functional hardware. Manufacturing new computers and smartphones is *very* energy- and resource-intensive – the production of chips, batteries, and components involves mining raw materials and global supply chains. Research suggests that the *emissions from manufacturing ICT equipment can be as large as – or even greater than – the emissions from using them*[31]. By pushing frequent upgrade cycles (often because the software became too heavy for the old hardware), we effectively multiply the carbon footprint of computing: first in making the new device, and then in powering it. Ethically, this is problematic because it's a form of planned obsolescence driven not by the users' needs, but by software inefficiency. A more efficient software ecosystem could **extend the useful life of devices**, reducing the churn of gadgets and the mountains of electronic waste (global e-waste is over 50 million tons per year and growing). From an environmental standpoint, writing efficient code is as important as building energy-efficient hardware; both are needed to curb ICT's footprint.

The numbers above underscore that **software efficiency is directly tied to sustainability**. Every wasted CPU cycle, every algorithm that runs in 100 steps when it could run in 10, ultimately translates into electricity drawn from the grid. When aggregated across millions of devices, inefficient software design can consume gigawatts of power and emit megatons of CO_2. Conversely, making software more efficient (sometimes called "green software" development) is a

powerful lever for environmental benefit. A simple initiative like optimizing a popular application to use 40% less CPU could **save enormous amounts of energy worldwide** if that app is running on, say, 500 million PCs. Indeed, some software projects explicitly advertise efficiency as a feature: the developer of the lightweight text editor Notepad++ famously stated that by optimizing routines and using efficient C++ code, *"Notepad++ is trying to reduce the world carbon dioxide emissions"*, because using less CPU power means lower energy draw and a "greener environment"[32]. It might sound grandiose for a text editor, but this statement recognizes a genuine truth: **efficient software is eco-friendly software**.

For policymakers and corporate leaders, these environmental findings mean that software choices and IT policies should be part of sustainability planning. It's not just a matter of what hardware or energy source you use, but *what code you choose to run*. Encouraging use of efficient software, avoiding overly bloated applications, and investing in optimization can be part of an organization's carbon reduction strategy. Just as "green building" design became a trend, **green computing and lean software** must become a priority. The next section will also show that it's not only about carbon and electrons – inefficient software has a very real human cost as well.

The Human Cost: Productivity and Time Wasted

In addition to environmental harm, inefficient computing causes a subtler but deeply personal kind of damage: it **steals time from people's lives**. In the aggregate, the effect is astonishing – billions of small delays add up to entire human lifetimes lost every year. From the office worker waiting for a sluggish spreadsheet to recalculate, to the casual user staring at a "Not Responding" program, these moments of delay and frustration have become so common that we accept them as normal. But should we? Time is a non-renewable resource, and designing software that unnecessarily wastes users' time is **an ethical failing in user-centric design**. Let's consider the scale of the issue:

- A study by University of Copenhagen and Roskilde University found that, on average, people **waste between 11% and 20% of their computer time dealing with delays and issues** – essentially time when "the system was slow, froze, or crashed" or the user was struggling with an unintuitive interface[33]. This study spanned a range of professions (students, office workers, IT professionals) and reflects general computing tasks. Up to one-fifth of computer usage time was effectively *lost* to system issues and inefficiencies[33]. Participants commonly reported problems like *"the system*

was slow" or *"it is difficult to find things"* as reasons for lost time[34]. Importantly, these weren't rare one-off glitches – **84% of the time, the same problems had occurred before** for the user, and nearly as often they expected the problem to recur[35]. This indicates persistent, systemic inefficiencies rather than freak incidents.

- Translate those percentages to a work week, and the results are eye-opening. Wasting 11–20% of your computer time could mean losing **almost a full day of work each week**. In fact, the researchers noted that it could be "half to a whole day of a normal working week" lost to computer troubles[36]. For a full-time employee, that's potentially hundreds of hours per year. For a large company, it's the equivalent of losing dozens of employee-years of productivity across the staff – essentially paying people to wait for progress bars or struggle with bloated software. One UK survey likewise revealed that the **average office worker loses about 24 working days per year** to slow or outdated technology[37] – that's actually more time than many workers get in annual vacation! The average employee in that survey lost 46 minutes per day to tech delays, which again comes out to roughly 8 hours a month, or 6–7 weeks per year of lost time[38]. When "time is money," the economic cost of this is huge – one analysis pegged it at around £3,000 per employee per year in the UK, factoring in wages paid for unproductive time[39]. But beyond money, think of the **human experience**: frustration, stress, and the demoralizing feeling of being held back by tools that are supposed to empower you.

- Specific examples of these delays are things we've all experienced. The **morning boot-up** of a computer can take several minutes on an older machine loaded with heavy startup applications – in the UK survey, just booting up each day added up to 8.8 days per year of waiting[40]. Opening large applications or files, dealing with an overloaded browser with too many scripts, or waiting for a search function to find a simple result – these are daily friction points. And while each instance may be only seconds or a minute, across billions of devices those seconds are a **collective tragedy of wasted human potential**. To illustrate: if a software update causes an extra 5 seconds delay in a common task for 1 billion users each day, that's 5 billion seconds lost daily – which is roughly **158 years of human time wasted per day**, globally. In a year, that single inefficiency would consume about 57,000 years of human time. No matter how small a delay seems, at scale it becomes enormous.

- Psychological and social impacts are also significant. Workers report that when technology "can't keep up," it causes them to lose focus and even feel less satisfied with their jobs[41]. Frustration with slow, unresponsive systems can lead to stress and burnout. In the survey, **1 in 10 employees said persistent tech problems made them want to quit their job**[42]. Over a quarter felt pressure to work overtime to compensate for tech-related delays[43], essentially donating their personal time to make up for inefficiencies. These human factors are often overlooked in IT budgeting. We spend on new features and new hardware, but not enough on ensuring the system responds *quickly* and *reliably* to the user. Yet from the user's perspective, **speed and responsiveness are foundational features**. A flashy new capability in an app means nothing if the app constantly hangs or lags.

The waste of human time due to inefficient software is ethically relevant because it shows **disrespect for the user**. People's time on this planet is precious; designing software that routinely squanders that time (especially in aggregate) is an implicit statement that the user's time is less important than other considerations (like developer convenience or added features). Good engineering and design should strive to **give time back to the user** – or at least not steal it needlessly. This is part of the imperative: efficiency in computing is not just about energy, but about dignity and respect for the end-user's life.

The Illusion of Progress: Inefficiency vs. Feature Gains

One might argue that modern software, despite being inefficient, delivers far more functionality than the lean software of the past – thus the inefficiency is the price of progress. However, on closer examination, there is **little correlation between how bloated software is and the value it delivers to the user**. In many cases, software has become inefficient *without* delivering substantially more value, or it delivers new features that could have been achieved without such high costs. This section dismantles the notion that inefficiency is acceptable as a byproduct of added features, and shows that true innovation does not inherently require greater resource usage.

Firstly, many *new features* are simply not relevant to a majority of users. For example, modern word processors and email clients are packed with advanced functions (like smart formatting, AI assistants, integration with cloud services, etc.), but studies and usage data often show that **most users utilize only a small core set of features**. In Microsoft Office, it's often cited that the average user taps into less than 10% of the available features. One analysis found that just five

commands (like Paste, Save, Copy, Undo, Bold) made up nearly a third of all commands used in Word[44]. The flip side is that *90% of the complexity is rarely touched by typical users*. Yet everyone pays the price in performance, because the software loads all that functionality and carries the weight of it. In effect, a vast portion of modern software's code is **dead weight** for most users' purposes, existing to check a box or serve niche cases while slowing down the common cases. The ethical question here: is it right to burden billions of users (and the planet) with the cost of features that only a tiny fraction need? A more efficient approach might be modular software – load features on demand – but that requires careful design that many products lack.

Secondly, **most inefficiencies are purely architectural and bring no user benefit at all**. The Windows 11 Start Menu example is emblematic: Microsoft reimplemented parts of the Start Menu using web-like technologies (React Native) [19], possibly to facilitate certain integrations or developer workflows. The end result was a Start Menu that cause *noticeable CPU spikes (30–70% on a core)* each time it opens[19], and users observed lag and stutters where none existed before. From a features perspective, the Start Menu didn't meaningfully gain from this – it simply shows apps and a "Recommended" file list with some cloud integration. All of that could be (and was) done with native code far more efficiently in earlier Windows versions. In this case, the inefficiency isn't tied to a "feature" that users asked for; it's a side effect of internal development choices. As one technologist lamented, *"The decay of the Start Menu into a laggy, unpredictable surface for advertising is perhaps the pinnacle of Windows' downfall"*[9] – pointing out that what should be a simple utility has turned into a vehicle for unnecessary web content and even ads, undermining the user experience. This is faux progress: a more complex implementation delivering a worse outcome.

The broader pattern is that **software often grows for reasons other than delivering core value**. Competitive pressures can lead companies to keep changing interfaces (sometimes for trendy aesthetics or "freshness" rather than true improvement), integrate new services (to capture users in ecosystems), or collect more data (telemetry and analytics running in the background). These changes can make software heavier without making it more *useful*. A candid comment from an observer of the software industry put it this way: *"Everything in Windows these days is wasting time stealing your data, loading ads or other unnecessary data from cloud services, and interacting with 'AI.' Performance is one of the lowest priorities... Microsoft consistently replaces things with modernized, but worse, versions and never returns to finish making the new version as good as the previous version that evolved over decades."*[45]. While a bit harsh, this critique

captures the feeling that **new does not always mean better** – especially if “new” means a reset of maturity. Often, mature efficient software is replaced with a “modern” rewrite that lacks polish and runs slower, just to fit a new business model or design trend.

This illusion of progress is dangerous because it normalizes inefficiency. Users come to believe that needing the latest hardware to run basic tasks is “just how it is,” or that waiting is part of computing. It doesn’t have to be. When software is thoughtfully engineered, feature advancement can happen *in tandem* with efficiency. An example is the video game industry: game developers operate under tight performance constraints (games must run at 60+ frames per second or users notice immediately) and yet have delivered astonishing improvements in graphics and gameplay. They achieve progress by optimization, innovating in algorithms, and using hardware acceleration wisely – not by accepting slowness. This shows that when performance is treated as a feature, progress doesn’t suffer; in fact, it excels. Unfortunately, in many general software domains, performance is not given the same priority. As a result, we have text editors (Electron-based) that use more RAM than an entire operating system from 20 years ago, or note-taking apps that consume as much CPU as an enterprise database.

It is also worth noting that **true innovation can sometimes *save* resources**. For example, modern compression algorithms, peer-to-peer distribution, or efficient cloud architectures can deliver new capabilities *while reducing* data transfer and storage needs. If similar inventive effort were put into everyday software efficiency as is put into adding new bells and whistles, we could break the cycle of bloat. The fact that some software (e.g., certain Linux distributions or lightweight apps) manage to stay extremely efficient even today indicates that inefficiency is not a requirement of modernity – it’s a byproduct of certain choices.

In ethical terms, delivering new features *without* considering efficiency is an **incomplete notion of progress**. Progress in computing should be measured not only by *what* software can do, but how *efficiently* it does it – achieving the goal without waste. The best technology improves peoples’ lives without collateral damage. Therefore, claiming that inefficient software is justified by its features ignores the possibility (and responsibility) to **innovate in smarter ways**. The nearly zero correlation between bloat and user benefit is a call to action: we must demand and develop software that advances on both functionality and efficiency axes together.

Towards Efficient and Sustainable Computing: A Call to Action

The evidence is clear that inefficient software is not just a harmless annoyance – it is a serious issue with ethical, environmental, and societal dimensions. We stand at a crossroads where we must consciously choose to reverse the trend of ever-expanding software bloat. The **ethical imperative** of efficient computing can be summed up simply: *we should not squander resources – whether electricity, hardware, or human lifetime – for want of care and discipline in software design.* The following are key takeaways and recommendations to drive this point home:

- **Recognize Efficiency as a Feature:** policymakers and industry leaders should treat software efficiency on par with security, accessibility, and other quality metrics. This means incorporating efficiency requirements into software projects (e.g., setting performance budgets, energy use targets) and spending the required development cost to achieve them. Just as energy labels exist for appliances, we may need **efficiency rating** for software. Imagine choosing a word processor not just on price or interface, but knowing which one gets the job done in the resource efficient way. Informed consumers can create demand for leaner software.

- **Incentivize Sustainable Software Development:** At the corporate and government level, incentives could be provided for software that demonstrably reduces carbon footprint. For example, governments could offer "green tech" certifications or tax benefits for software companies that optimize their applications to extend device lifetimes (thus reducing e-waste) and minimize energy use. Large tech firms and cloud providers can lead by optimizing their own services – every CPU cycle saved in a data center is magnified by scale. The **Green Software Foundation** and similar initiatives are already bringing companies together to share best practices for building energy-efficient applications[46]. This momentum needs to continue and expand.

- **Educate and Empower Developers:** Many software engineers have never been trained to think about energy efficiency or taught the old-school optimization techniques that were necessary in a bygone era. Most modern software development consists of importing huge amounts of prefabricated software parts "libraries" into the code, then writing a modest amount of code to glue the prefabricata together in ways that achieve the desired outcome, this makes development easier and often much faster, but it introduces potentially enormous amounts of resource waste. Incorporating

principles of **“green code” and efficient algorithms** into computer science curricula and developer training can raise awareness. Simple practices like algorithmic optimization, including efficiency when considering whether to create something specifically for the problem at hand or use prefabricata, and choosing efficient data structures can have large impacts. Moreover, developers should be given tools to measure the performance and energy impact of their code. Profiling and monitoring should include not just speed but power consumption. With the right feedback, engineering teams can iteratively improve efficiency just as they do with speed or memory leaks. Much of the energy usage a developer can cause comes not from the code that developer wrote, but from the libraries and frameworks they used.

- **Embrace “Digital Sufficiency” Philosophy:** A concept emerging in sustainability circles is *digital sufficiency* – using the right amount of tech, but not more than necessary[47]. In practical terms for software, this means avoiding over-engineering. If a lightweight solution exists (e.g., a native app or a simpler framework), consider using it instead of a heavy, generalized one, especially for simple tasks. It also means reconsidering feature sets: pursue the **80/20 rule** – focus on the 20% of features that deliver 80% of value, and implement them in the most efficient way possible. Features that are rarely used but add considerable bloat might be made optional or left out, especially if they trigger continuous resource usage.
- **Optimize for Longevity:** Efficient computing aligns with **right to repair and device longevity** movements. Software should be developed to **run well on older hardware** for as long as possible. Not everyone has the latest device, nor should they need it to perform basic tasks. By making software scalable (able to turn off fancy effects or choose lower resource modes), developers can ensure inclusion and reduce the forced-upgrade cycle. This could involve releasing *“light” or “classic” versions* of apps aimed at low-end hardware or simply incorporating adaptive performance settings. It’s encouraging that some software makers have started doing this (for instance, offering HTML lite versions of web email for slow connections, or “battery saver” modes that reduce animations and background activity).
- **Policy and Cultural Change:** At the organizational level, IT policy can discourage the deployment of needlessly heavy applications. Governments and enterprises that procure software can ask vendors to provide information on resource requirements and efficiency, and favor software that meets efficiency benchmarks. Culturally, we can also combat the notion

that *"newer is automatically better."* Users and managers should question updates that degrade performance. If an update makes the user experience worse (slower, more bloated) without clear benefit, that should be seen as a regression – perhaps even an unacceptable one. This mindset shift can pressure software providers to optimize rather than assume customers will just buy new hardware.

- **Measure and Publicize Impact:** We need more research and transparency on the impact of software inefficiency. Just as we have statistics on how many tons of CO_2 a flight produces, we should quantify what a heavy app or inefficient code costs in emissions. Some researchers have proposed a **Software Carbon Intensity (SCI) metric** to rate applications by how much carbon they emit per unit of work[48][49]. If such metrics were widely published, it could influence procurement and consumer choices. Additionally, publicizing the *human time* lost to slow software in terms of aggregate years could be a powerful moral statistic that grabs headlines. For instance, imagine a yearly "Global Software Waste Report" that says "X million human-hours and Y GWh of energy were lost due to software inefficiencies this year." This kind of report can spur action in the same way reports on traffic congestion or air pollution do.

In conclusion, **efficient computing is an ethical imperative** because it speaks to how we steward both our planet's resources and our own time. The status quo of bloated software is not only technically suboptimal; it is ethically unsound when viewed in the light of climate change and respect for users. We have the knowledge and the tools to do better. As the adage goes, *"With great power comes great responsibility."* Our computers and software today have *great power* – far more than necessary for most tasks. Therefore, it is our responsibility to harness that power judiciously, avoiding wanton waste. By re-aligning priorities – valuing optimization, acknowledging the hidden costs of inefficiency, and striving for lean design – the tech industry can innovate in a way that is sustainable and respectful. The progress of tomorrow should not be measured just in **features added**, but also in **waste eliminated**. The ethical choice is clear: we must make efficiency a cornerstone of computing's future, so that technological advancement and environmental stewardship go hand in hand, and so that computing empowers humanity *without* inadvertently hindering it through waste.

Sources:

- Niklaus Wirth, *A Plea for Lean Software*, on software bloat vs hardware progress[6][4].
- University of Copenhagen study on time wasted due to computer problems[33][36].
- Belkhir & Elmeligi (2018) on ICT's rising share of global carbon footprint[21].
- Manner et al. (2022) on energy unsustainability of software, "software bloat...aggravates problems"[13].
- Notepad++ project statement on optimizing for lower CO2 emissions[32].
- Pawprint eco analysis on carbon footprint of emails (0.3g CO_2 per email on average)[26].
- Currys/Elite study: 24 days/year lost by office workers to slow tech[37].
- Hacker News discussion highlighting Windows 11 Start Menu inefficiencies[50][9] and Electron app memory usage vs native[14].
- Jeff Atwood, *Software is a Gas* (citing Nathan Myhrvold's laws of software growth)[51].
- HTTP Archive data on web page bloat, ~3-12× growth in page sizes over a decade[22].
- Comments on modern Windows focusing on ads/telemetry at expense of performance[45].

(All citations are provided to underscore claims and statistics made in this paper. The urgency for efficient computing is backed by these multidisciplinary findings, from academic research to industry surveys and expert observations.)[21][11][32][33]

---

[1] [3] [4] [5] [6] Wirth's law - Wikipedia

https://en.wikipedia.org/wiki/Wirth%27s_law

[2] [11] [13] [17] [18] [22] [23] [24] [25] [31] [46] (PDF) Black software - the energy unsustainability of software systems in the 21st Century

https://www.researchgate.net/publication/366626189_Black_software_-_the_energy_unsustainability_of_software_systems_in_the_21st_Century

[7] [8] [9] [10] [50] Windows 11 Start Menu Revealed as Resource-Heavy React Native App | Hacker News

https://news.ycombinator.com/item?id=44124688

[12] [15] [16] [51] Software: It’s a Gas

https://blog.codinghorror.com/software-its-a-gas/

[14] Any Chance of a Native Non-electron Desktop app? - Password Manager - Bitwarden Community Forums

https://community.bitwarden.com/t/any-chance-of-a-native-non-electron-desktop-app/16854

[19] Windows 11 Start Menu Revealed as Resource-Heavy React Native ...

https://winaero.com/windows-11-start-menu-revealed-as-resource-heavy-react-native-app-sparks-performance-concerns/

[20] [45] Windows 11 Start Menu Revealed as Resource-Heavy React Native ...

https://lemmy.ca/post/45020319/16835617

[21] Assessing ICT global emissions footprint: Trends to 2040 & recommendations | Health & Environmental Research Online (HERO) | US EPA

https://hero.epa.gov/hero/index.cfm/reference/details/reference_id/7696418

[26] [27] [29] [30] What’s the carbon footprint of an email?

https://www.pawprint.eco/eco-blog/carbon-footprint-email

[28] The Carbon Cost of an Email: Update! - The Carbon Literacy Project

https://carbonliteracy.com/the-carbon-cost-of-an-email/

[32] Article - What is Notepad++?

https://msudenver.teamdynamix.com/TDClient/2313/Portal/KB/ArticleDet?ID=144133

[33] [34] [35] [36] Tech News : Study Shows 20% Of Time Wasted Within IT - Surf Tech IT

https://surftechit.co.uk/tech-news-study-shows-20-of-time-wasted-within-it/

[37] [38] [39] [40] [41] [42] [43] Office workers waste more time on slow tech than they spend on holiday - Elite Business Magazine

https://elitebusinessmagazine.co.uk/people/item/office-workers-waste-more-time-on-slow-tech-than-they-spend-on-holiday

[44] The Most Frequently Used Features in Microsoft Office

http://googlesystem.blogspot.com/2008/02/most-frequently-used-features-in.html

[47] [49] Digital sufficiency: conceptual considerations for ICTs on a finite planet

https://pmc.ncbi.nlm.nih.gov/articles/PMC10427517/

[48] How Green Is Your Software? - Harvard Business Review

https://hbr.org/2020/09/how-green-is-your-software