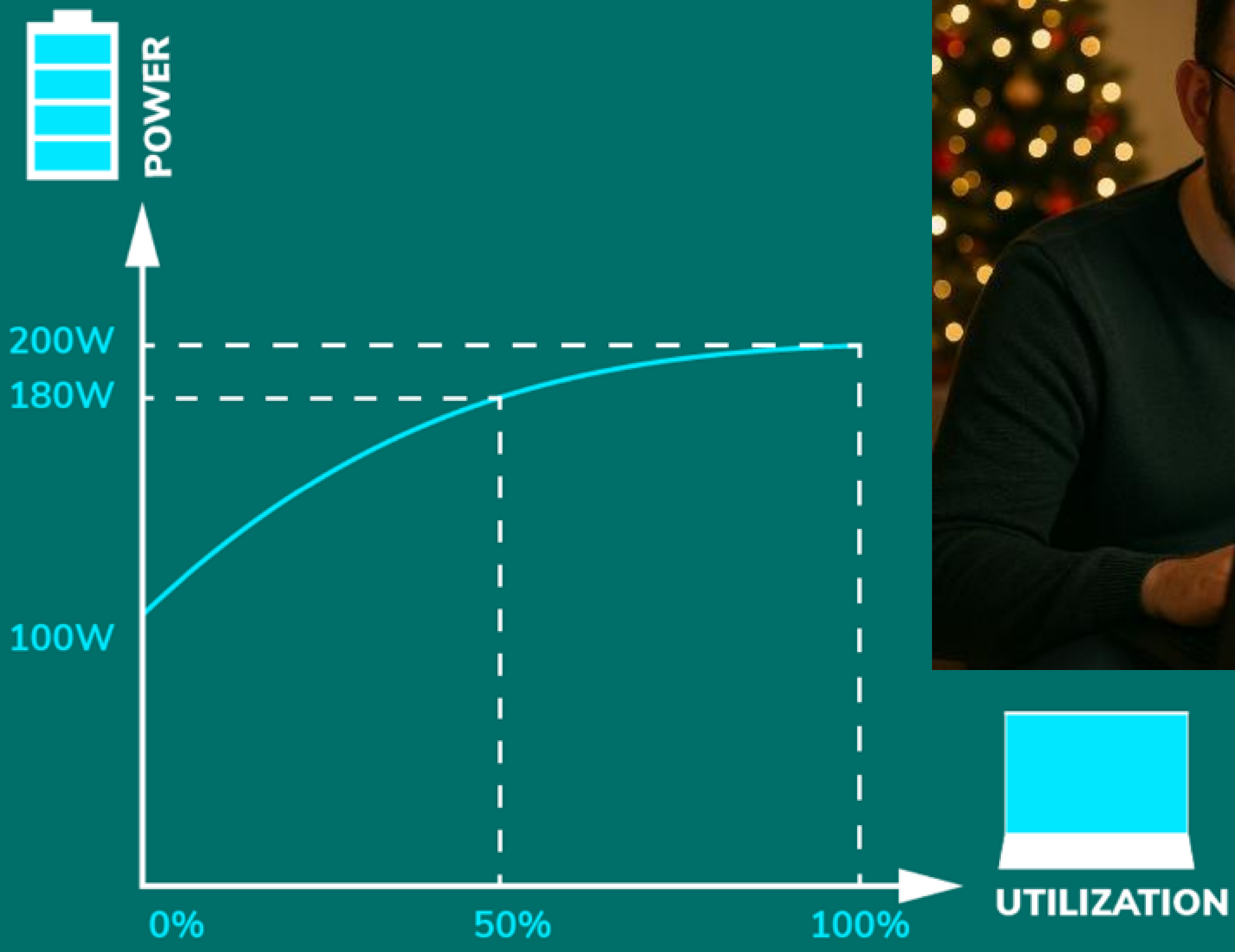

Your Software & AI
emits tonnes of CO_2.
You can cut it.
Want to know how?
HighTech Innovators
YOUR SOFTWARE, BETTER

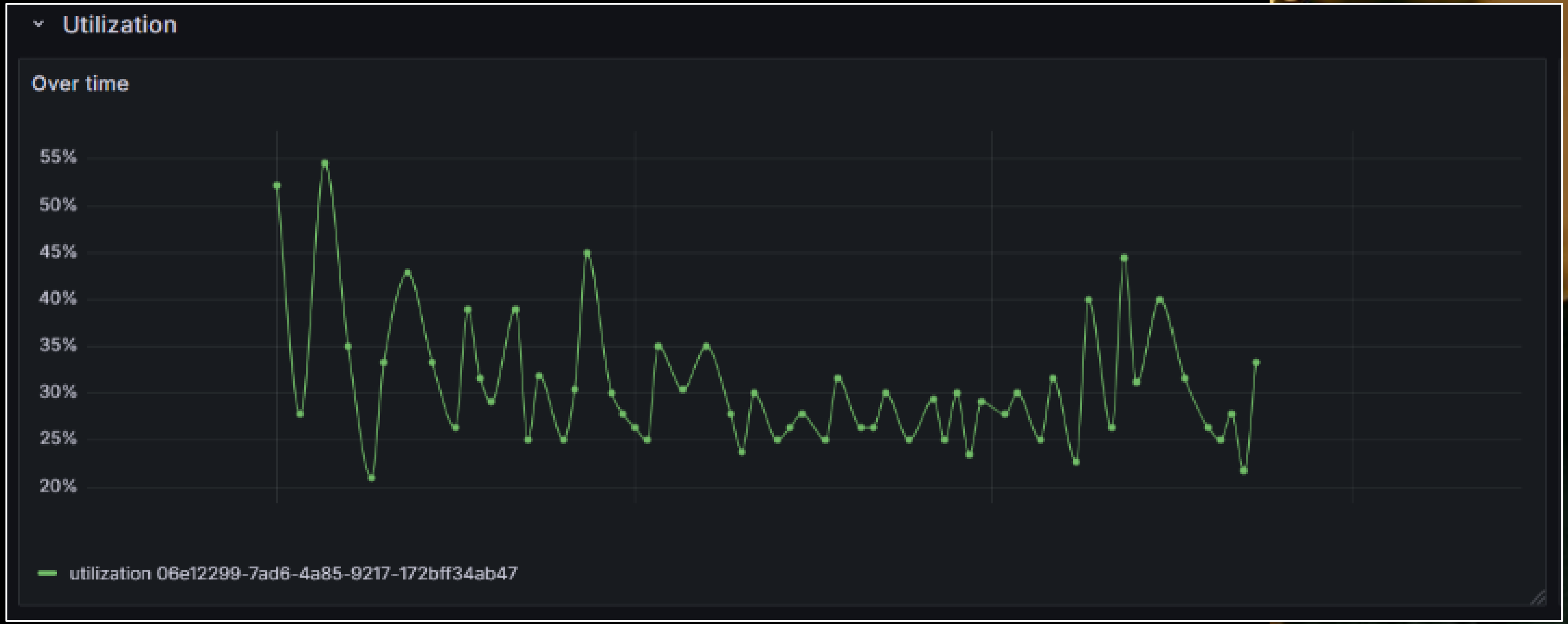
Green Software CHAMPIONS

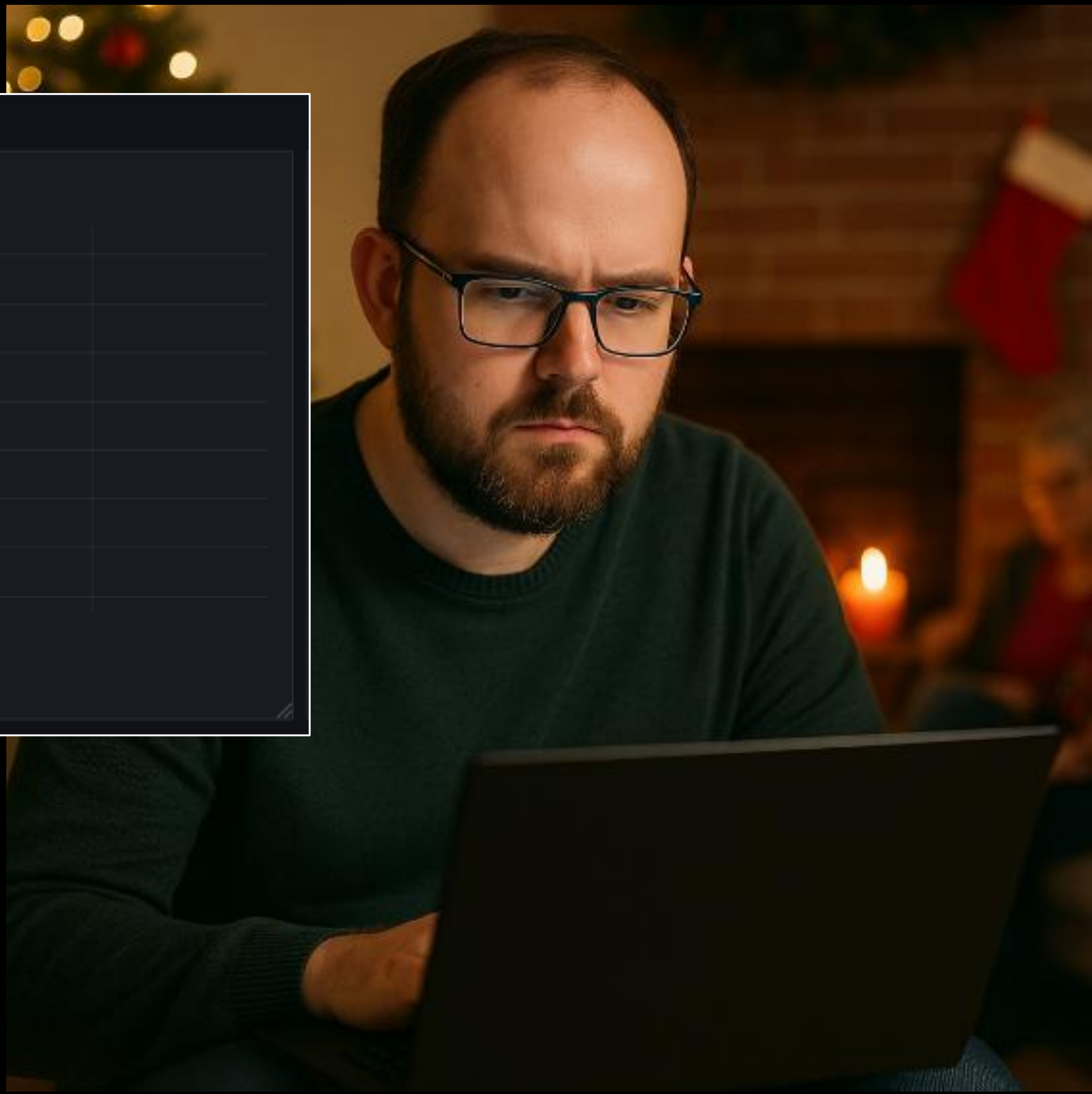
Green Software Foundation

It’s Christmas 2024…

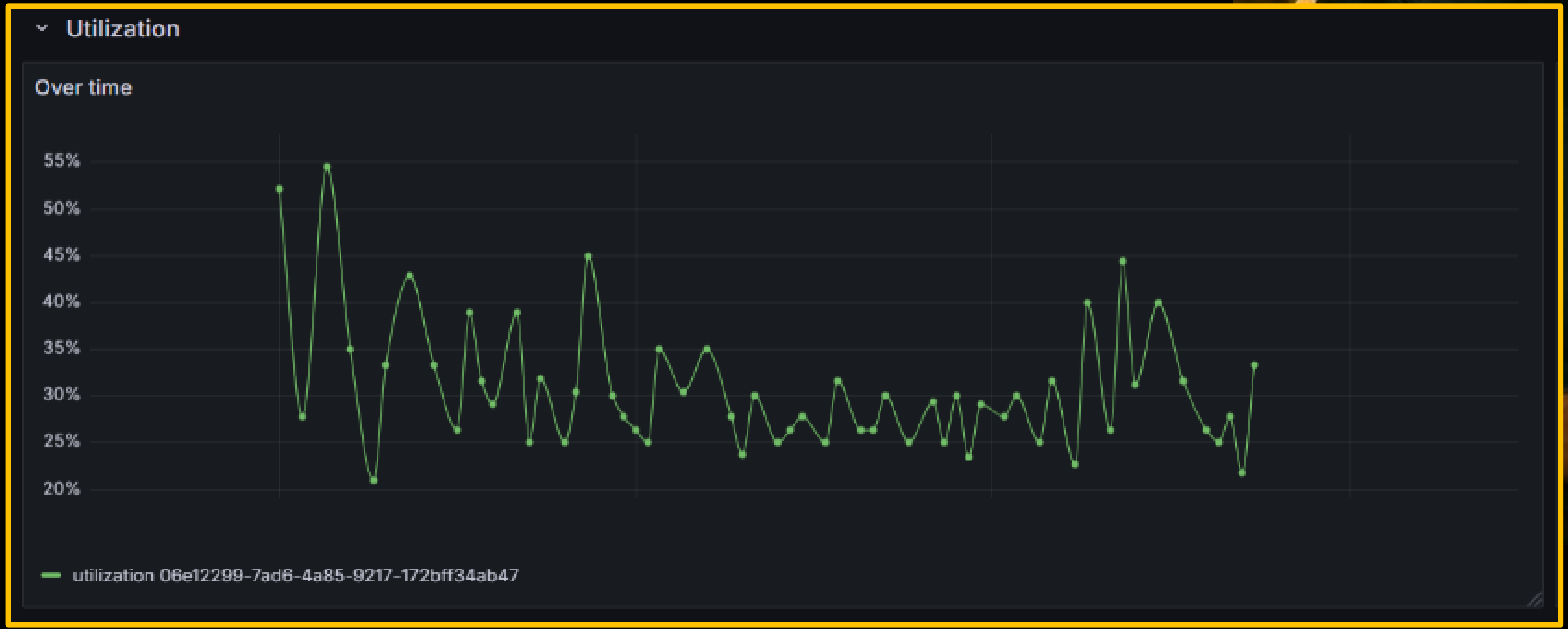


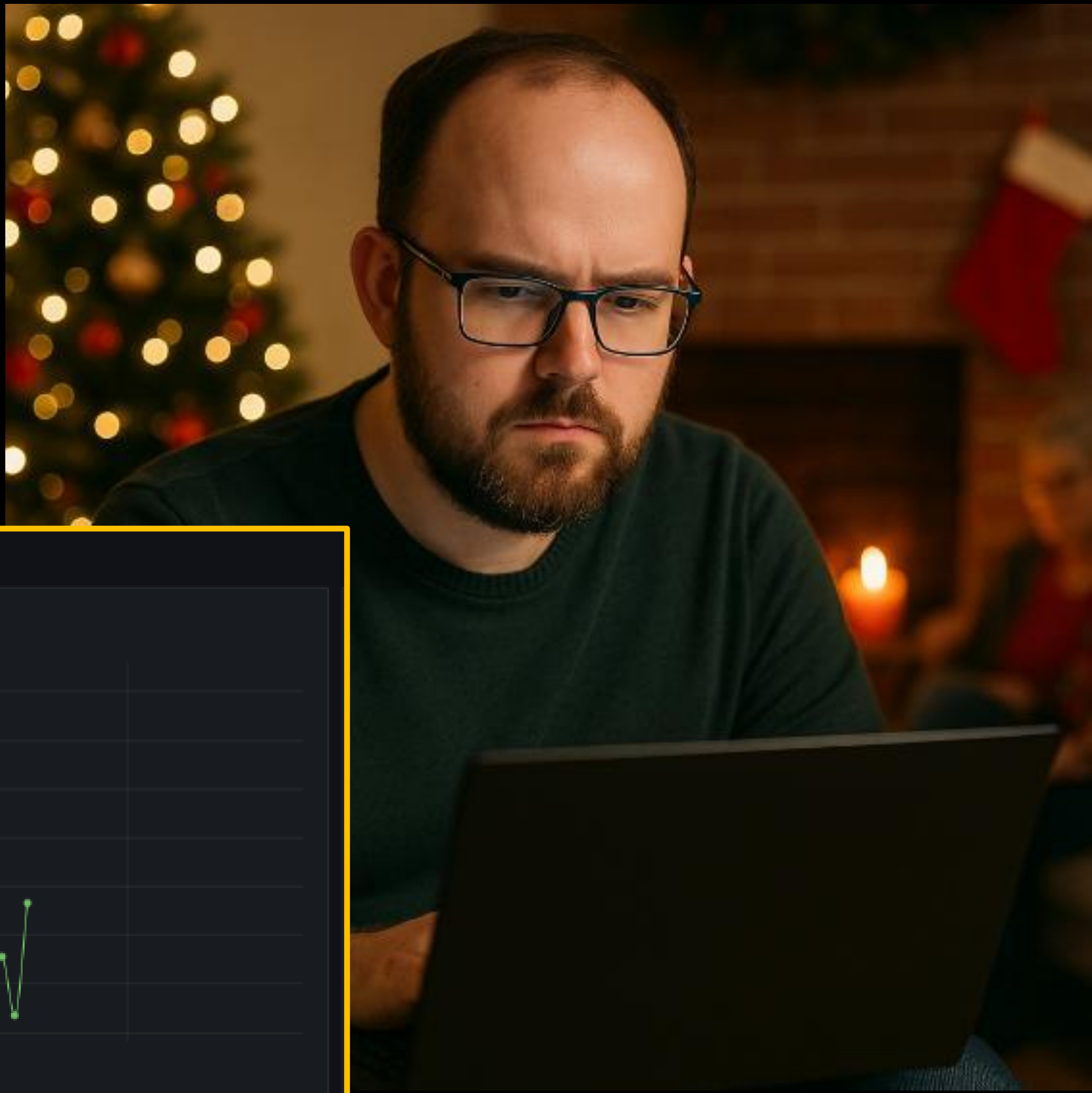
Let's revisit a lesson from LFC 131.
Was it correct and complete?

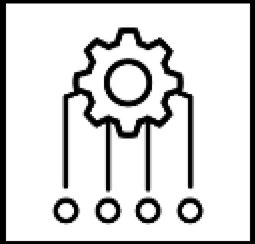

Utilization
Over time
55%
50%
45%
40%
35%
30%
25%
20%
utilization 06e12299-7ad6-4a85-9217-172bff34ab47



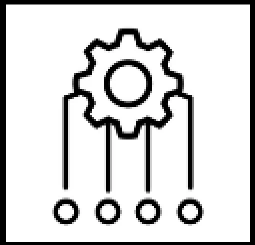

Utilization
Over time
100%
80%
60%
40%
20%


Utilization
Over time
55%
50%
45%
40%
35%
30%
25%
20%
utilization 06e12299-7ad6-4a85-9217-172bff34ab47






AMD
RYZEN

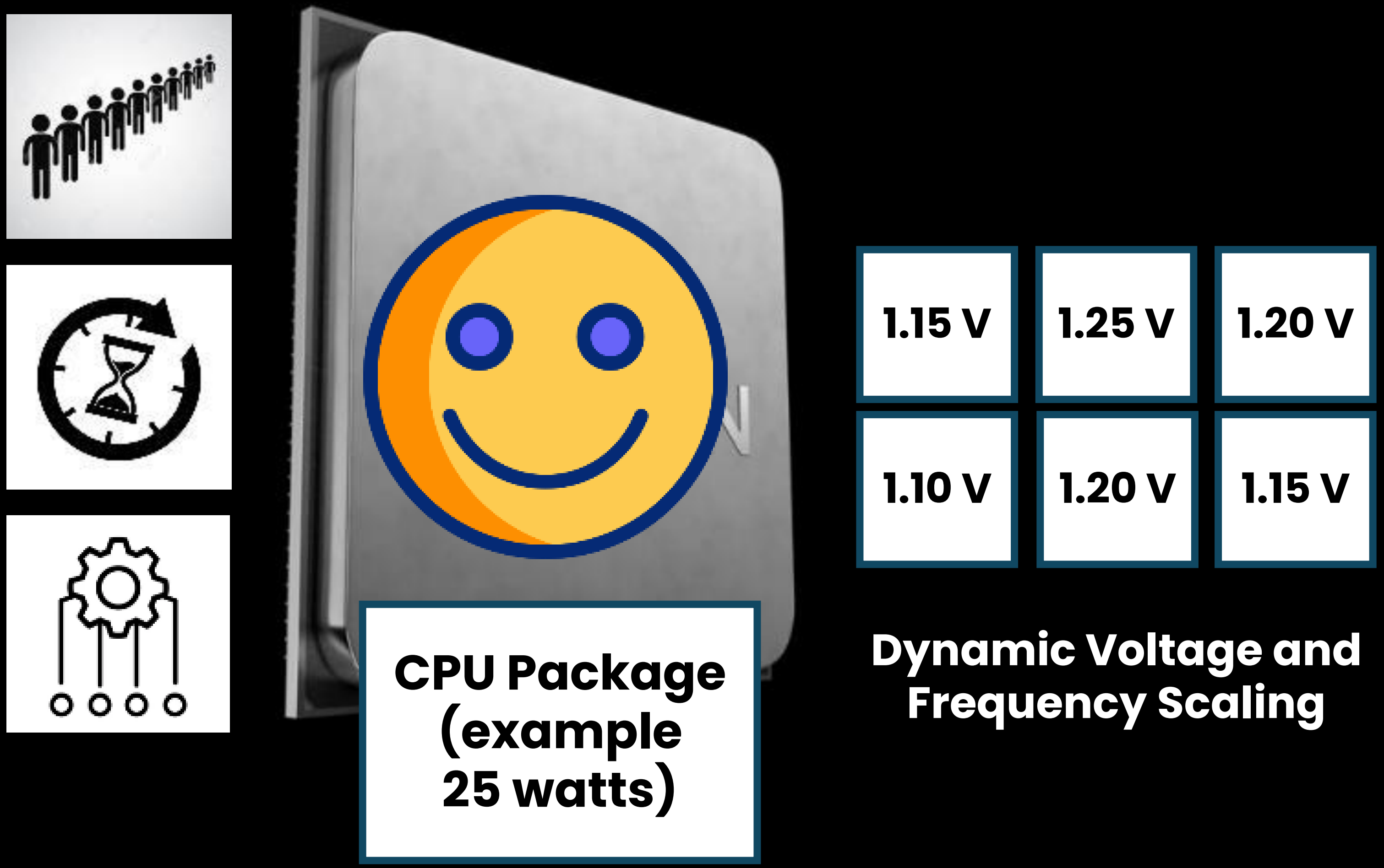


5 Ghz	5 Ghz	Idle
Idle	Idle	Idle

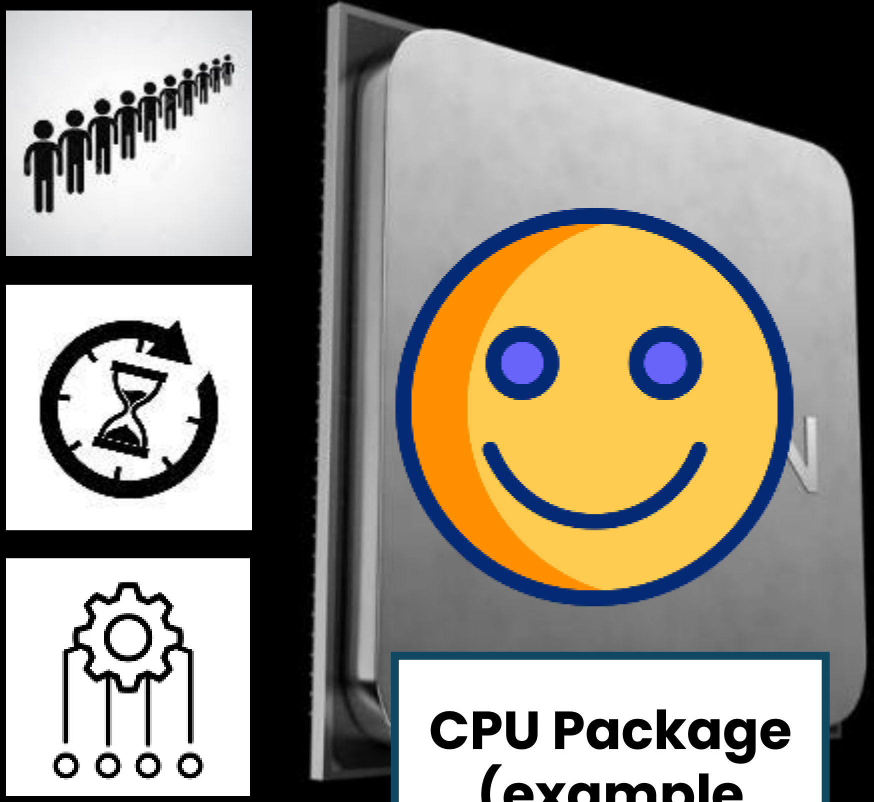

5 Ghz
5 Ghz
5 Ghz
5 Ghz
5 Ghz
5 Ghz

3.2 Ghz	3.6 Ghz	3.4 Ghz
3.0 Ghz	3.4 Ghz	3.2 Ghz

Dynamic Voltage and Frequency Scaling



1.15 V
1.25 V
1.20 V
1.10 V
1.20 V
1.15 V
CPU Package (example 25 watts)
Dynamic Voltage and Frequency Scaling



CPU Package (example 25 watts)

1.12 mWh	1.26 mWh	1.19 mWh
1.05 mWh	1.19 mWh	1.12 mWh

Over 1 second

Designing for Flow,
Not Just Execution

Synchronous / Blocking

Tasks queue up the CPU waits between operations

Looks simple, but causes *idle energy draw* and *lower throughput*

Asynchronous / Flow-Oriented

Tasks overlap and share resources efficiently

Keeps systems *active, responsive, and energy-smart*

Poor Design Choices

⬇

Underutilization

⬇

Energy Waste + Cost + Latency

Built to Last

Your software can run **100% on renewables** and still be **unsustainable.**

💚 *You’re already greener than you think.*

- Running on renewable grids
- Optimizing for performance & cost
- Writing efficient code
- Automating scaling & provisioning
- Reducing idle compute

But sustainability starts *where efficiency ends.*

Green Design vs Sustainable Design

Does this
line of *code*, this *build*,
or this *instance*
create **lasting value**,
or just more activity?

Meet ParallelQuickSort

```
using System;
using System.Threading.Tasks;

public static class ParallelQuickSort
{
    public static void Sort<T>(T[] a) where T : IComparable<T> => Sort(a, 0, a.Length - 1);

    static void Sort<T>(T[] a, int l, int r) where T : IComparable<T>
    {
        if (l >= r) return;
        int i = l, j = r; T p = a[(l + r) / 2];
        while (i <= j)
        {
            while (a[i].CompareTo(p) < 0) i++;
            while (a[j].CompareTo(p) > 0) j--;
            if (i <= j) { (a[i], a[j]) = (a[j], a[i]); i++; j--; }
        }
        if (r - l < 10000)
        {
            if (l < j) Sort(a, l, j);
            if (i < r) Sort(a, i, r);
        }
        else Parallel.Invoke(
            () => { if (l < j) Sort(a, l, j); },
            () => { if (i < r) Sort(a, i, r); }
        );
    }
}
```

≈ 140–160 million uOps

```
using System;
using System.Threading.Tasks;

public static class ParallelQuickSort
{
    public static void Sort<T>(T[] a) where T : IComparable<T> => Sort(a, 0, a.Length - 1);

    static void Sort<T>(T[] a, int l, int r) where T : IComparable<T>
    {
        if (l >= r) return;
        int i = l, j = r; T p = a[(l + r) / 2];
        while (i <= j)
        {
            while (a[i].CompareTo(p) < 0) i++;
            while (a[j].CompareTo(p) > 0) j--;
            if (i <= j) { (a[i], a[j]) = (a[j], a[i]); i++; j--; }
        }
        if (r - l < 10000)
        {
            if (l < j) Sort(a, l, j);
            if (i < r) Sort(a, i, r);
        }
        else Parallel.Invoke(
            () => { if (l < j) Sort(a, l, j); },
            () => { if (i < r) Sort(a, i, r); }
        );
    }
}
```

```csharp
using System;
using System.Threading.Tasks;

public static class ParallelQuickSort
{
    public static void Sort<T>(T[] a) where T : IComparable<T> => Sort(a, 0, a.Length - 1);

    static void Sort<T>(T[] a, int l, int r) where T : IComparable<T>
    {
        if (l >= r) return;
        int i = l, j = r; T p = a[(l + r) / 2];
        while (i <= j)
        {
            while (a[i].CompareTo(p) < 0) i++;
            while (a[j].CompareTo(p) > 0) j--;
            if (i <= j) { (a[i], a[j]) = (a[j], a[i]); i++; j--; }
        }
        if (r - l < 10000)
        {
            if (l < j) Sort(a, l, j);
            if (i < r) Sort(a, i, r);
        }
        else Parallel.Invoke(
            () => { if (l < j) Sort(a, l, j); },
            () => { if (i < r) Sort(a, i, r); }
        );
    }
}
```

VS

```csharp
list.Sort();
```

```
list.Sort();
```

So, it performs roughly the same *total work* as your ParallelQuickSort, but it’s usually **faster in wall time.**

≈ 120–160 million uOps

```
list.Sort();
```

```csharp
using System;
using System.Threading.Tasks;

public static class ParallelQuickSort
{
    public static void Sort<T>(T[] a) where T : IComparable<T> => Sort(a, 0, a.Length - 1);

    static void Sort<T>(T[] a, int l, int r) where T : IComparable<T>
    {
        if (l >= r) return;
        int i = l, j = r; T p = a[(l + r) / 2];
        while (i <= j)
        {
            while (a[i].CompareTo(p) < 0) i++;
            while (a[j].CompareTo(p) > 0) j--;
            if (i <= j) { (a[i], a[j]) = (a[j], a[i]); i++; j--; }
        }
        if (r - l < 10000)
        {
            if (l < j) Sort(a, l, j);
            if (i < r) Sort(a, i, r);
        }
        else Parallel.Invoke(
            () => { if (l < j) Sort(a, l, j); },
            () => { if (i < r) Sort(a, i, r); }
        );
    }
}
```

VS

```csharp
list.Sort();
```

```
// Ascending sort by property "Prop1".
MagicSorter.Sort(ref list, "Prop1", SortType.Asc);
```

VS

```
list.Sort();
```

```
// Ascending sort by property "Prop1".
MagicSorter.Sort(ref list, "Prop1", SortType.Asc);
```

About

A wide-use sorting library for .NET Core.

- Readme
- MIT license
- Activity
- 0 stars
- 1 watching
- 0 forks

Report repository

VS

```
list.Sort();
```

```
// Sequential file downloads
foreach (var url in urls)
{
    var data = await new HttpClient().GetStringAsync(url);
    Process(data);
}
```

VS

```
// Parallel async downloads with controlled concurrency
var tasks = urls.Select(url => GetAndProcessAsync(url));
await Task.WhenAll(tasks);
```

```
// Sequential file downloads
foreach (var url in urls)
{
    var data = await new HttpClient().GetStringAsync(url);
    Process(data);
}
```

UNDERUTILIZATION

```
// Parallel async downloads with controlled concurrency
var tasks = urls.Select(url => GetAndProcessAsync(url));
await Task.WhenAll(tasks);
```

rust

```rust
use tokio::task;

#[tokio::main(flavor = "multi_thread", worker_threads = 8)]
async fn main() {
    let tasks: Vec<_> = (0..100_000)
        .map(|_| task::spawn(async { 42 }))
        .collect();
    for t in tasks { t.await.unwrap(); }
}
```

✅ 3–5 uOps per await
✅ Full hardware concurrency
✅ ≈ 0.0004 mWh / 100 k ops

⚙️ ~40 uOps per await
⚙️ ≈ 0.0015 mWh / 100 k ops
🔴 *Interpreter & IPC cost dominate energy.*

python

```python
import asyncio, concurrent.futures

async def work(): return 42

async def main():
    loop = asyncio.get_running_loop()
    with concurrent.futures.ProcessPoolExecutor() as p:
        tasks = [loop.run_in_executor(p, work) for _ in range(100_000)]
        await asyncio.gather(*tasks)

asyncio.run(main())
```

Rust's async is hard because you must manage who owns what, instead of the system doing it for you.

But does **this** make developers **avoid async development**?

The unseen

Most of the energy waste in software doesn't happen in algorithms, it happens in **how systems idle, talk, and scale.**

Why do we design systems to be **always on**?

What are
we afraid will happen
if they rest?

Let's take this example VM

Running at 30% utilization

~180 W system draw,

including PUE.

0.18 kWh per hour

Let’s take this example VM

Running at 30% utilization

~180 W system draw,

including PUE.

4.32 kWh per day

Let’s take this example VM

Running at 30% utilization

~180 W system draw,

including PUE.

≈ 131 kWh per month

Let’s take this example VM

Running at 30% utilization

~180 W system draw,

including PUE.

≈ 1576 kWh per year

Let’s take this example VM

Running at 75% utilization

~270 W system draw, including PUE.

≈ 2365 kWh per year

1000 VMs

Running at 75% utilization

~270 W system draw, including PUE.

≈ 2,365,200 kWh per year

1000 VMs

Running at 75% utilization

~270 W system draw,

220 g CO_2e / kWh (location-based method)

520 metric tons CO_2e per year

15000 VMs

Running at 75% utilization

~270 W system draw,

220 g CO_2e / kWh (location-based method)

7.8 kilotons CO_2e per year

41,600,000,000,000,000 grams CO_2e

global CO_2 emissions per year (IPCC/Global Carbon Project)

41,600,000,000,000,000 grams CO_2e

global CO_2 emissions per year (IPCC/Global Carbon Project)

1,886,098,000,000 grams CO_2e

Scope 3 Bechtle AG emissions in 2024

15,801 employees → 22,804,000,000 grams CO_2e

Scope 1 & 2 Bechtle AG emissions in 2024

520,300,000 grams CO_2e

1000 virtual example servers (220g co2e / kWh)

220 g CO_2e/kWh

2023 CBS

ARCHITECTURE

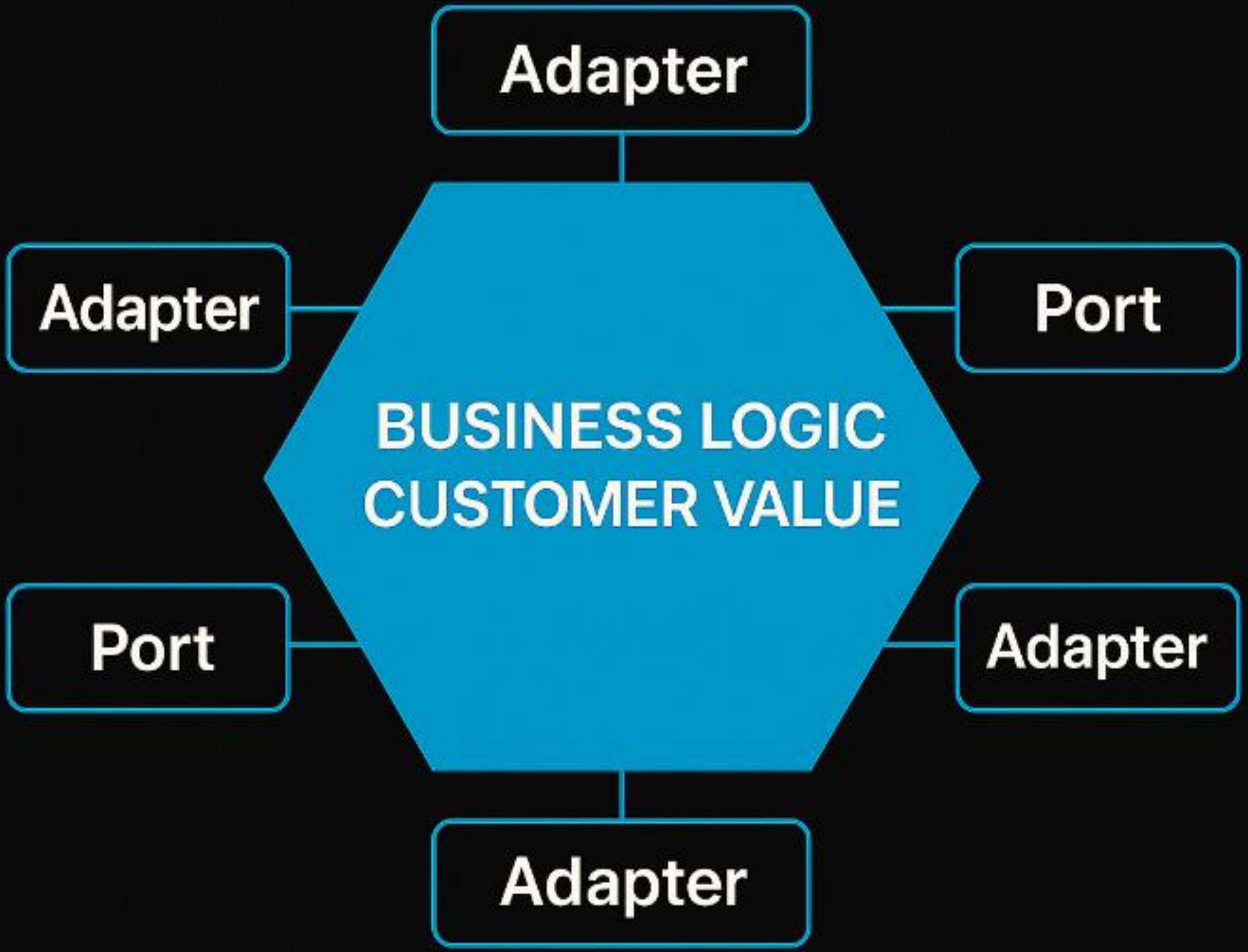
Sustainability begins with
how we think, plan, and build.

4 Context **Environments**

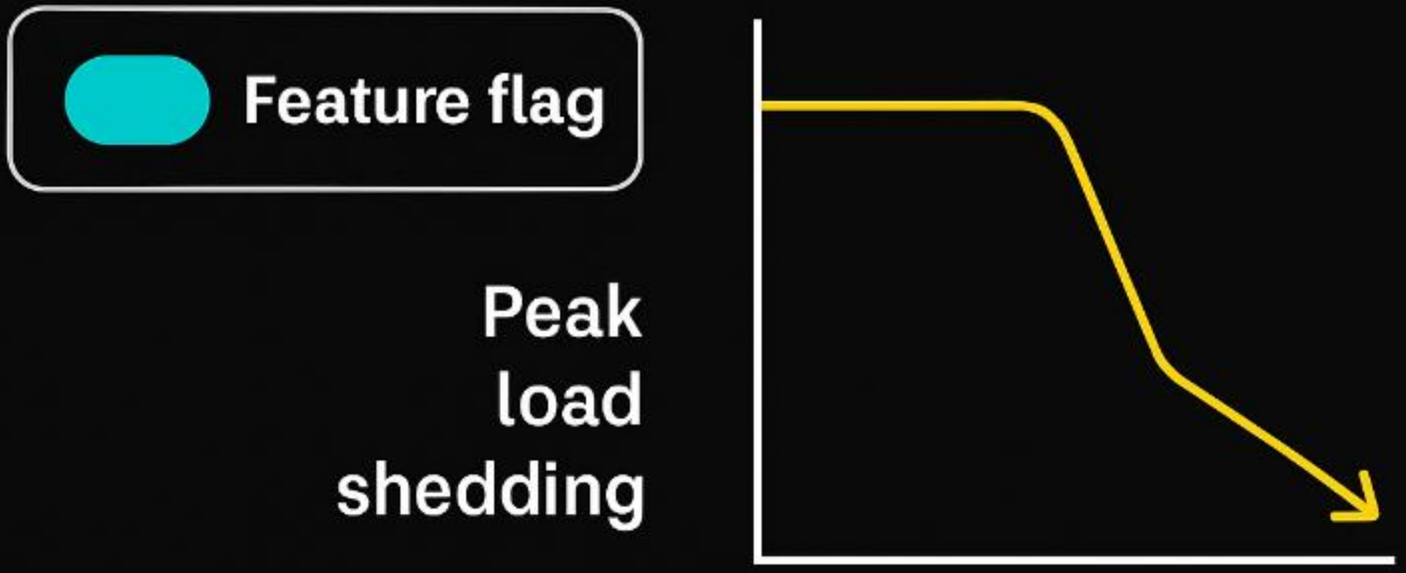
- ☁️ **Cloud**
- 📦 **On-Prem**
- 🧳 **Traveling**
- 🏠 **@Home**

So, everything serverless?

Value as the Bedrock



Graceful Degradation and Peak Load Mitigation



METRICS

Measuring What Matters

Core Philosophy

Energy and CO_2 are the *truth check*, **not the *target*.**

Everyone expect me now
to go to energy and co2e and
Software Carbon Intensity.
Right?

Every efficient system starts with a hypothesis.

If we ***improve X****, we should see lower* ***waste at Y****.* Metrics exist to confirm that hypothesis.

From Detection → Measured Impact

Development

Docker Build
(Code + Image)

Test

Docker Image
CI/CD Tests

Acceptance

Argo CD Deploy
to Acceptance

Production

Argo CD Sync
to Prod Cluster

Detect
Underutilization
Bottlenecks
Waiting Patterns

Validate
Reduction in
kWh and Co2

The DevOps++ pillars

Code Changes	IaC Changes	Architecture Repository Updated	Less Energy & Co2 In Production

MODELS

Rethinking Energy in Code

Let’s design with awareness, knowing how many **VMs**, **containers**, and **instances** are running somewhere on hardware.

Let’s design with awareness, **how many tokens** did it take to push that **PR**?

Design is decision-making

Theoretical model of
@ 1 Ghz, 1 Hz == 1 uOps,
@ 1 Ghz @ 1 core == 1.2 Volt.
And respect Dynamic Voltage and
Frequency Scaling

GreenR - Reasoning

Theoretical model of
@ 1 Ghz, 1 Hz == 1 uOps,
@ 1 Ghz @ 1 core == 1.2 Volt.
And respect Dynamic Voltage and Frequency Scaling

GreenR - Reasoning

Thought for less than a second

Theoretical Model Overview

PARAMETER	SYMBOL	TYPICAL VALUE (BASELINE)
Clock frequency	f	1 GHz
Micro-operations per second	u	$u = f$ (1 Hz → 1 μOp)
Core supply voltage	V	1.2 V at 1 GHz
Dynamic power (per core)	P_{dyn}	$P_{\text{dyn}} = \alpha C V^2 f$
Static (leakage) power	P_{leak}	≈ $I_{\text{leak}} V$ (temperature-dependent)

Key assumptions

1. **Linear μOp-frequency mapping** – each clock tick can retire at most one μOp, so the μOp rate equals the clock frequency ($u = f$).
2. **Voltage-frequency relationship** – to keep the silicon timing safe, the supply voltage must rise roughly with the square-root of frequency (empirically $V \propto \sqrt{f}$ for many modern cores). The baseline point is $f_0 = 1\,\text{GHz}, V_0 = 1.2\,\text{V}$.
3. **Dynamic power scaling** – follows the classic CMOS equation $P_{\text{dyn}} = \alpha C V^2 f$ where α is the activity factor and C the effective switched capacitance per cycle.

1 Frequency-Voltage Curve (DVFS)

Send a message

+ Zoeken op het web   Deep Search

GreenR - Reasoning

Inklappen Opslaan Kopieer

```
using System.Linq;                               // no runtime energy

class Program                                    // no runtime energy
{
    static void Main()
    {
        const int N = 1_000_000;                 // compile-time constant – no runtime energy
        var rnd = new Random();                  // 5 µOp  → 2 e-9 mWh ≈ 0.000002 mWh
        var list = new List<int>(capacity: N); // 10 µOp → 4 e-9 mWh = 0.000004 mWh

        // Loop body: 21 µOp per iteration → 21 µOp × 1 000 000 = **21 000 000 µOp**
        for (int i = 0; i < N; i++)             // loop overhead 3 µOp/iter → 0.0012 mWh
        {
            int a = rnd.Next(1000), b = rnd.Next(1000);
            // 10 µOp/iter (two RNG calls) → 0.004 mWh

            int sum = a + b, prod = a * b;       // 3 µOp/iter → 0.0012 mWh
            bool even = (sum % 2 == 0);          // 3 µOp/iter → 0.0012 mWh

            list.Add(even ? prod : sum);         // 2 µOp/iter → 0.0008 mWh
        }                                        // **Loop total ≈ 0.0084 mWh**

        list.Sort();                             // ≈200 M µOp (Timsort) → 0.080 mWh
        var groups = list.GroupBy(x => x);       // ≈1.08 M µOp (hash buckets + scan) → 0.000432 mWh
        var maxFreq = groups.Max(g => g.Count());// ≈1 M µOp (final scan) → 0.000400 mWh

        Console.WriteLine($"Max frequency: {maxFreq}");
        // 30 µOp → 1.2 e-8 mWh (negligible)

        // ----------------------------------------------------------------
```

Send a message

+ Zoeken op het web Deep Search

Last remark

Sustainability should be achievable for every developer, not just those on clean grids.

Closing & Q&A

When code, people, and purpose align, technology becomes sustainable by nature.

Pretty print

{"power_usage_J_per_ms":0.0085,"power_usage_W":8.5,"timestamp":"2025-05-13 05:25:01.658"}