The Joule Story

A Language Born from Necessity

David Jean Charlot, PhD

Open Interface Engineering, Inc. (openIE)

david@openie.dev

February 2026 | Version 1.0

Executive Summary

Joule is a systems programming language designed for energy-efficient computing. Born from the recognition that software's energy consumption has become a critical environmental concern, Joule provides developers with tools to build high-performance applications while minimizing their carbon footprint. This document tells the story of why Joule was created and the principles that guide its development.

---

1. The Context: Computing's Growing Footprint

In the early 2020s, a troubling reality became impossible to ignore: the world's digital infrastructure was consuming electricity at an unprecedented and accelerating rate.

1.1 The Numbers

The International Energy Agency reports that data centers alone consumed **240-340 TWh** of electricity in 2022, with projections reaching **1,000 TWh by 2026** as AI workloads proliferate [1]. To put this in perspective:

- Data centers now consume **more electricity than many countries** [2]
- A single large AI model training run can emit **300+ tonnes of CO2** [3]
- Global ICT could account for **up to 8% of electricity demand by 2030** [4]

–

The software industry had long operated under an implicit assumption: hardware would get faster, electricity would remain cheap, and efficiency was someone else's problem. This assumption was proving catastrophically wrong.

1.2 The Overlooked Variable

Much attention focused on hardware efficiency: better chips, improved cooling, renewable power procurement. These are important. But a critical variable remained largely unexamined: **the software itself**.

Research began revealing uncomfortable truths. A 2017 study by Pereira et al. demonstrated that the same algorithm, solving the same problem, could consume **75x more energy** when written in Python versus C [5]. The choice of programming language, a decision made early and rarely revisited, had massive downstream energy implications.

---

2. The Founding Insight

Joule began with a simple but profound observation: **the energy a program consumes is not an accident. It's a consequence of design decisions made at the language level.**

2.1 Language Design Matters

When a language forces you to use garbage collection, you accept unpredictable energy spikes during collection cycles. Research shows GC can account for **5–25% of program energy**, depending on allocation patterns and heap size [6].

When a language lacks support for heterogeneous computing, you leave GPU and specialized hardware efficiency on the table. TPUs can be **30–80x more energy-efficient** than CPUs for machine learning workloads [7].

When a language provides no visibility into energy consumption, developers fly blind, optimizing for metrics that may not align with environmental impact.

2.2 The Question

We asked: *What if we designed a language from first principles with energy efficiency as a core requirement, not a retrofit?*

–

This wasn't about creating yet another systems language. It was about recognizing that the environmental crisis demands new tools, tools that make sustainable computing not just possible, but natural.

A note on Rust: Rust achieves excellent energy efficiency through its ownership model, and we acknowledge its influence on Joule's memory management. However, Rust's primary design goal is memory safety, not energy awareness. Joule differs by making energy a first-class language concern (with built-in budgets, measurement, and thermal awareness) and by providing native heterogeneous computing support for GPUs, TPUs, and emerging AI accelerators. For developers whose primary constraint is energy rather than safety, Joule offers purpose-built tooling.

---

3. The Name

We chose the name "Joule" deliberately. James Prescott Joule (1818-1889) was the physicist who established the mechanical equivalent of heat—demonstrating that energy is conserved, that work and heat are interchangeable forms of the same fundamental quantity [8].

His work laid the foundation for the first law of thermodynamics and revolutionized our understanding of energy. The SI unit of energy bears his name.

In software, we had forgotten Joule's lesson. We pretended that computational work was free, that wasted cycles had no consequence. Every CPU instruction consumes energy. Every memory access generates heat. Every inefficient algorithm draws power that must come from somewhere.

Joule the language exists to remind us that **every operation has a cost, and that cost matters**.

---

4. Design Principles

Joule was built on three foundational principles, each addressing a specific dimension of energy efficiency.

4.1 Energy Awareness as a First-Class Concern

In most languages, you can measure execution time. In Joule, you can also express and measure energy.

–

```
@energy_budget(100.microjoules)
fn process_sensor_reading(data: &SensorData) -> ProcessedData {
    // Compiler estimates energy bounds; warns if budget may be exceeded
    ...
}
```

Our type system includes energy annotations, allowing developers to specify energy budgets and have the compiler check compliance through static analysis. This isn't about restriction. It's about **visibility and intentionality**.

4.2 Zero-Cost Abstractions That Actually Cost Zero

Many languages promise "zero-cost abstractions" but deliver hidden overhead. Joule's ownership and borrowing system, drawing on advances in type theory and memory management research [9], eliminates garbage collection entirely.

Memory is managed deterministically, with predictable, minimal energy overhead. The compiler proves memory safety at compile time, incurring **zero runtime cost** for safety guarantees.

Research demonstrates that languages with ownership-based memory management (like Rust) achieve energy efficiency comparable to manually-managed C, while providing strong safety guarantees [5].

4.3 Heterogeneous Computing by Default

Modern devices contain multiple processing units: CPUs, GPUs, TPUs, NPUs. Most languages treat these as exotic hardware requiring special libraries and expert knowledge.

Joule treats heterogeneous computing as normal. Writing code that runs efficiently across different processors should be as natural as writing a function:

```
@kernel
fn matrix_multiply(a: &Matrix, b: &Matrix) -> Matrix {
    // Implementation is portable across CPU/GPU/TPU
    ...
}
```

The language runtime, guided by hardware telemetry, can route computations to the most energy-efficient available processor.

–

5. Technical Foundations

5.1 Compiler Architecture

Joule employs a tri-backend compiler architecture:

Backend	Use Case	Characteristics
Cranelift	Development	Fast compilation for rapid iteration
LLVM	Production	Maximum optimization, broad platform support
MLIR	Emerging Hardware	AI accelerators, custom silicon

This flexibility ensures Joule can target current hardware optimally while adapting to the evolving landscape of specialized processors.

5.2 Energy-Aware Optimization

The compiler performs optimizations specifically targeting energy consumption [10]:

- **Memory traffic minimization**: DRAM access consumes ~100x more energy than cache access
- **SIMD vectorization**: More computation per instruction fetch
- **Power state enablement**: Avoiding patterns that prevent processor sleep
- **Code density**: Better instruction cache utilization

5.3 Hardware Integration

Joule integrates with hardware power monitoring interfaces like Intel RAPL and ARM Energy Probe [11], providing developers with actual energy measurements rather than estimates.

---

6. What Joule Offers

For developers concerned about the environmental impact of their work, Joule provides:

- **Transparency**: See where your program's energy goes
- **Control**: Set energy budgets and have the compiler help you meet them
- **Efficiency**: Write code that does more with less power
- **Future-readiness**: Target emerging energy-efficient hardware without rewriting your codebase

- **Safety**: Memory safety without garbage collection overhead

---

7. Looking Forward

The software industry stands at a crossroads. We can continue treating energy as an externality, pushing the costs onto the planet and future generations. Or we can acknowledge that sustainable computing requires sustainable tools.

Joule is our contribution to the latter path. It's not the only answer, but it's one answer. A demonstration that programming languages can and should take energy seriously.

The International Energy Agency projects that without significant efficiency improvements, data center energy consumption could **triple by 2030** [1]. The choices we make today about how software is built will shape that trajectory.

---

8. Conclusion

The name Joule carries another meaning for us. It's a unit of work. And there's much work to be done.

We invite you to join us in building software that respects the physical realities of our world, software that performs its function while minimizing its footprint on a planet we all share.

References

[1] International Energy Agency. "Data Centres and Data Transmission Networks." IEA, 2024. https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks

[2] International Energy Agency. "Electricity 2024: Analysis and Forecast to 2026." IEA, January 2024. https://www.iea.org/reports/electricity-2024

[3] Patterson, D., Gonzalez, J., Le, Q., et al. "Carbon Emissions and Large Neural Network Training." arXiv:2104.10350, 2021. https://arxiv.org/abs/2104.10350

[4] Andrae, A.S.G., Edler, T. "On Global Electricity Usage of Communication Technology: Trends to 2030." Challenges, vol. 6, no. 1, pp. 117-157, 2015. https://doi.org/10.3390/challe6010117

–

[5] Pereira, R., Couto, M., Ribeiro, F., et al. “Energy Efficiency across Programming Languages.” Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE), 2017. https://joule.openie.dev/research/energy-efficiency-languages

[6] Pinto, G., Castor, F., Liu, Y.D. “Mining Questions About Software Energy Consumption.” Proceedings of the 11th Working Conference on Mining Software Repositories (MSR), 2014. https://doi.org/10.1145/2597073.2597110

[7] Jouppi, N.P., Young, C., Patil, N., et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit.” Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), 2017. https://doi.org/10.1145/3079856.3080246

[8] Joule, J.P. “On the Mechanical Equivalent of Heat.” Philosophical Transactions of the Royal Society of London, vol. 140, pp. 61-82, 1850. https://doi.org/10.1098/rstl.1850.0004

[9] Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D. “RustBelt: Securing the Foundations of the Rust Programming Language.” Proceedings of the ACM on Programming Languages, vol. 2, no. POPL, 2018. https://doi.org/10.1145/3158154

[10] Schulte, E., Dorn, J., Harding, S., et al. “Post-compiler Software Optimization for Reducing Energy.” Proceedings of ASPLOS, 2014. https://doi.org/10.1145/2541940.2541980

[11] *Khan, K.N., Hirki, M., Niemi, T., et al. “RAPL in Action: Experiences in Using RAPL for Power Measurements.” ACM TOMPECS, vol. 3, no. 2, 2018. https://doi.org/10.1145/3177754*

–