CodeGreen: Towards Improving Precision and Portability in Software Energy Measurement

Saurabhsingh Rajput, Tushar Sharma
Dalhousie University
Halifax, Canada
{saurabh,tushar}@dal.ca

Abstract

Accurate software energy measurement is critical for optimizing energy, yet existing profilers force a trade-off between measurement accuracy and overhead due to tight coupling with supported specific hardware or languages. We present **CodeGreen**, a modular energy measurement platform that decouples instrumentation from measurement via an asynchronous producer-consumer architecture. We implement a Native Energy Measurement Backend (NEMB) that polls hardware sensors (Intel RAPL, NVIDIA NVML, AMD ROCm) independently, while lightweight timestamp markers enable tunable granularity. CodeGreen leverages Tree-sitter AST queries for automated instrumentation across Python, C++, C, and Java, with straightforward extension to any Tree-sitter-supported grammar, enabling developers to target specific scopes (loops, methods, classes) without manual intervention. Validation against *Computer Language Benchmarks Game* demonstrates $R^2 = 0.9934$ correlation with RAPL ground truth and $R^2 = 0.9997$ energy-workload linearity. By bridging fine-grained measurement precision with cross-platform portability, CodeGreen enables practical algorithmic energy optimization across heterogeneous environments. Source code [22], video demonstration [19], and documentation [18] for the tool are publicly available.

Keywords

Energy Measurement, Power Profiling, Green Software

1 Introduction

The cessation of Dennard scaling [7] and the surge in energy-intensive AI workloads [31] have transformed software energy consumption into a critical constraint on operational costs and environmental sustainability. While the "Green Software" [26] movement posits that code itself—its logic, algorithms, and architectural patterns—is a primary driver of carbon emissions, the practical optimization of these software artifacts faces a critical tooling gap: *measuring energy consumption at source-code granularity without compromising measurement validity.* Over the past decade, substantial progress has been made in developing energy measurement tools, yet the majority remain tailored for research environments, requiring specialized hardware access, manual instrumentation, and domain-specific expertise. These barriers impede widespread adoption among software developers, who are the primary agents of energy optimization at the algorithmic and architectural levels.

Existing tools often presume homogeneous execution environments, targeting specific hardware (*e.g.,* Intel RAPL [15]) or specific languages (*e.g.,* Python frameworks). However, modern software development is characterized by extreme heterogeneity: a codebase may execute across diverse hardware platforms (Intel, AMD, NVIDIA GPUs, ARM processors) while hosting applications written in multiple languages (Python, C++, Java). A developer-centric energy profiling tool must therefore be agnostic to hardware and programming language. Simultaneously, developers seeking to identify energy hotspots during iterative development require fine-grained visibility into individual APIs and methods. This necessity amplifies a fundamental tension in software metrology: the "*Granularity-Overhead*" trade-off *i.e.,* as measurement targets smaller execution units (individual functions or loops), the overhead of the measurement instrument grows disproportionately, rendering data statistically invalid. Therefore, an energy measurement tool must deliver high measurement accuracy while minimizing noise and overhead.

Current tools exhibit critical limitations. HPC-grade profilers (PAPI [27], Likwid [24], EnergAt [14]) achieve microsecond resolution but require root access and manual instrumentation. ML-focused tools (CodeCarbon [6], Eco2AI [4]) prioritize ease of use but sacrifice accuracy through coarse 15-second sampling and TDP heuristics. Cloud-native tools such as Kepler *et al.* [1] have negligible overhead but operate at process granularity, lacking function-level visibility. Language-specific profilers (PyRapl [9]) provide fine-grained attribution but incur prohibitive overhead from synchronous hardware reads and runtime contention. Hardware support remains fragmented: fine-grained profilers couple tightly to vendor interfaces (Intel RAPL), while GPU tools lack unified CPU correlation, forcing developers to manage disparate toolchains. These tools reveal a consistent gap: developer-focused profiling requires high temporal resolution and control over granularity without measurement artifacts, automated instrumentation, low and predictable overhead, and multi-hardware and multi-language capability.

To address these limitations, we present **CodeGreen**, a modular energy measurement platform architected for hardware and language extensibility. CodeGreen separates instrumentation from measurement: applications record lightweight timestamp markers via Tree-sitter-based [3] AST analysis, enabling seamless language extension. The **Native Energy Measurement Backend (NEMB)** asynchronously polls diverse hardware sensors (Intel RAPL, NVIDIA NVML, AMD ROCm) through an extensible driver interface, enabling precise multi-language profiling across heterogeneous environments. Validation demonstrates $R^2 = 0.9934$ correlation with RAPL ground truth and $R^2 = 0.9997$ energy-workload linearity. CodeGreen is available as an open-source toolkit [22] with documentation [18], targeting developers optimizing algorithmic energy consumption across diverse deployment environments.

2 Background and Related Work

The Physics of Software Energy. Software energy profiling maps logical program execution to physical power consumption. Unlike

intrinsic metrics such as execution time, energy (E) is a physical quantity derived from the integral of instantaneous power (P) consumed over time: $E = \int_{t_0}^{t_1} P(t)\,dt$. Digital power consumption is a continuous signal driven by transistor switching. To reliably reconstruct this signal, the sampling frequency must satisfy the *Nyquist-Shannon theorem* [21], mandating that the sampling rate $f_s > 2f_{\max}$, where $f_{\max}$ represents the highest frequency component of the power waveform. However, hardware sensors impose physical ceilings; RAPL updates at ~1 kHz (1 ms intervals). Profilers that sample significantly slower than this hardware limit (*e.g.,* CodeCarbon's default 15s) are mathematically guaranteed to *alias* [28] the signal, missing transient power spikes caused by bursty workloads.

The Observer Effect. Measurement introduces computational perturbations to the system under observation. Synchronous profilers (*e.g.,* pyRAPL, pyJoules) halt application threads to read hardware registers via kernel context switches. This flushes instruction pipelines, evicts cache data, and prevents low-power CPU states. For fine-grained measurements, this overhead can dominate execution time, invalidating energy attribution.

Instrumentation Strategies. *Manual instrumentation* (*e.g.,* PMT [5], Likwid [24]) requires developers to explicitly annotate code regions, imposing high cognitive burden and placement error risk. *Automated instrumentation* uses compiler or AST analysis to inject checkpoints programmatically [20]. However, naive automation can lead to over-instrumentation, resulting in unexpected measurement overheads that distort workload behavior. Effective instrumentation balances coverage against intrusion.

Hardware Interface Landscape. *CPU Telemetry.* Intel RAPL exposes cumulative energy counters for Package, Cores, and DRAM domains. While RAPL provides reasonable resolution,it suffers from counter wraparound under high load and security restrictions (*e.g.,* post-Platypus attacks), forcing tools to use slower filesystem interfaces. AMD and ARM offer similar capabilities via `amd_energy` and `hwmon`, but with varying granularity and update frequencies. *GPU Telemetry.* NVIDIA NVML provides GPU power monitoring but exhibits significant limitations. Yang et al. [30] demonstrate that slow averaging windows (1s on consumer GPUs) and default polling rates miss up to 75% of kernel execution behavior. PowerSensor3 [25] confirms NVML consistently underestimates energy.

2.1 Comparative Tool Analysis

Existing tools present distinct trade-offs between precision, overhead, and usability.

HPC-Grade Profiling. PAPI [27] and Likwid [24] achieve low-latency profiling via direct hardware access but require root privileges and manual annotation. PMT [5] abstracts heterogeneous accelerators but incurs instrumentation overhead due to a lack of automation support across languages and granularity.

Python/ML Ecosystem. CodeCarbon [6] prioritizes ease of use with coarse 15s sampling intervals. Fischer *et al.* [11] demonstrate this underestimates energy by 20–40% for bursty workloads due to peripheral power exclusion and TDP heuristics. pyJoules [2], and pyRAPL [9] offer decorator-based profiling but suffers from synchronous I/O and GIL contention. Rajput *et al.* [20] provide energy measurement limited to Deep Learning Framework APIs.

Language-Specific Agents. JoularJX [16] provides Java-specific profiling via agent-based stack sampling but introduces up to 8% overhead and ±5% attribution error [23], as it infers energy from CPU demand rather than direct measurement.

Model-based Attribution. SmartWatts [10] correlates hardware performance counters with energy consumption but requires calibration times of minutes to days per platform [9] and mandates a complex stack (MongoDB, InfluxDB, sensor daemon), which is impractical for fine-granular profiling.

Infrastructure Middleware. Kepler [1] and Scaphandre [17] use ratio models—energy scales linearly with CPU utilization— suitable for billing and monitoring, but lacks method-level resolution.

3 Design and Implementation

CodeGreen differentiates itself through a design philosophy centered on hardware and language extensibility while minimizing measurement noise through asynchronous decoupling and predictable overhead modeling.

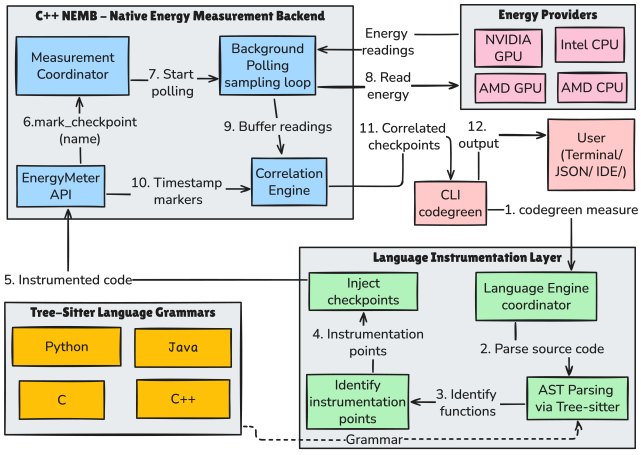

Figure 1: CodeGreen architecture

Asynchronous Producer-Consumer Architecture. The core architectural principle of CodeGreen is the **asynchronous decoupling** of the application under measurement (Producer) from the energy measurement subsystem (Consumer), illustrated in Figure 1. Drawing on event-driven architectures [8], CodeGreen eliminates the temporal dependency between execution and measurement. Traditional profilers entangle these concerns by blocking the application thread while synchronously querying hardware sensors. In contrast, CodeGreen's application thread performs only a lightweight operation: writing a timestamp marker to a lock-free, thread-safe circular buffer, enabling thousands of checkpoints per second without perturbing the application's energy profile.

The measurement workflow proceeds in three phases, as steps shown in Fig. 1: (1) **Instrumentation:** The Language Engine parses source code using Tree-sitter, identifies instrumentation points via SCM queries (steps 2-3), and injects checkpoint calls (step 4) that invoke the `EnergyMeter` API—the runtime interface responsible for registering execution boundaries. During execution, each checkpoint records only a high-resolution timestamp and unique identifier (step 6), incurring minimal overhead. (2) **Background Polling:**

A dedicated C++ thread continuously polls hardware sensors at *configurable intervals* (default 10ms, step 7-9), storing (`timestamp`, `cumulative_energy`) tuples in a lock-free buffer. This thread operates independently of the application, preventing measurement operations from blocking execution. (3) **Correlation:** Post-execution, the Correlation Engine (step 10-11) uses binary search and linear interpolation to map asynchronous checkpoints onto the energy time series, attributing consumption to code regions.

Hardware and Language Extensibility CodeGreen implements a modular driver architecture in the C++ backend (Figure 1, Energy Providers). The system defines a generic `EnergyProvider` interface abstracting vendor-specific hardware APIs. Integrating new platforms (*e.g.,* AMD ROCm, ARM energy counters) requires only implementing a driver class conforming to this interface without modifying core measurement logic. Current drivers include: **Intel/AMD RAPL** accessing CPU energy counters via Model Specific Registers (MSR) or Linux `perf`, supporting Package, Core (PP0), and DRAM domains; **NVML** monitoring GPU power via the NVIDIA Management Library, reporting Total Graphics Power with domain-level breakdown. The Measurement Coordinator(Figure 1) orchestrates multi-provider sampling, enabling simultaneous CPU and GPU profiling with synchronized timestamps.

Language support leverages the Language Instrumentation Layer (bottom of Figure 1), which uses Tree-sitter parsers to analyze source code via pattern-matching queries (SCM queries). The Language Engine coordinates instrumentation across Python, C, C++, and Java by invoking language-specific Tree-sitter grammars, identifying instrumentation points (*e.g.,* functions, loops, classes), and injecting checkpoint calls. Extending to additional languages requires only providing the corresponding Tree-sitter grammar. The measurement backend exposes a unified `EnergyMeter` API that language bindings invoke to mark checkpoints, ensuring consistent behavior across runtimes. Critically, instrumentation granularity is fully configurable: users define targeted scopes—from API endpoints to specific internal loops—via configuration files, precisely calibrating profiling depth to optimization targets.

Checkpointing and Thread Safety To support complex execution patterns including recursion and multithreading, each checkpoint uses a composite key: `function_name#inv_N_tTHREADID`, where `inv_N` distinguishes recursive invocations and `tTHREADID` isolates concurrent threads. This lightweight scheme enables dense instrumentation at function boundaries, loop iterations, and critical sections while maintaining thread safety without locks.

Predictable Overhead and Noise Reduction Measurement accuracy depends on both sensor precision and attribution correctness. CodeGreen enforces temporal consistency by timestamping all events—both application checkpoints and hardware sensor readings—using `CLOCK_MONOTONIC` [12], a monotonically increasing system clock immune to administrative adjustments or NTP synchronization [29]. This unified time reference enables the Correlation Engine to precisely align execution markers with energy measurements, attributing energy to specific code blocks even when hardware sensor update intervals (typically 1ms for RAPL) exceed individual function execution times.

Critically, the continuous polling strategy ensures predictable, constant overhead independent of application behavior. This deterministic behavior enables overhead normalization: since both the fixed initialization cost (T_{base}) and per-checkpoint cost ($T_{\text{checkpoint}}$) are constant for a given runtime, instrumentation overhead can be precisely subtracted from measurements using the model: $\text{Overhead}_{\text{norm}} = (T_{\text{inst}} - T_{\text{native}} - T_{\text{base}} - N \times T_{\text{checkpoint}})/T_{\text{native}} \times 100\%$, where N is the checkpoint count. This normalization yields true measurement overhead after accounting for instrumentation artifacts. Users can thus predict and remove instrumentation costs when interpreting energy measurements, obtaining energy consumption that closely approximates uninstrumented execution.

Data Correlation and Output Hardware sensor updates occur at fixed intervals (typically 1ms for RAPL) that rarely align with checkpoint timestamps. The Correlation Engine addresses this through temporal interpolation: given energy readings (t_i, E_i) and (t_{i+1}, E_{i+1}) and a checkpoint at time t_c where $t_i < t_c < t_{i+1}$, energy is estimated as $E(t_c) = E_i + (E_{i+1} - E_i) \cdot \frac{t_c - t_i}{t_{i+1} - t_i}$. This linear interpolation accurately captures cumulative energy for steady-state workloads. Users configure measurement granularity via CLI (`codegreen measure –interval 1ms`), trading temporal resolution for storage overhead. Output is structured as JSON (Listing 1), providing per-checkpoint energy attribution, aggregate statistics, and raw time-series data for external analysis.

Listing 1: CodeGreen hierarchical energy attribution output.

```
{
   "timestamp": "2026-01-16T14:30:00.123456",
      "analysis": {
         "language": "python",
         "success": true,
         "instrumentation_points_count": 5,
      },
      "measurement": {
         "total_energy_joules": 1.234,
         "total_time_seconds": 0.567,
         "checkpoints": [
            {   "checkpoint_id": "my_func#inv_1_t123",
                "type": "function_entry",
                "energy_joules": 0.001,
                "time_seconds": 0.0001 },
            {   "checkpoint_id": "loop_body#inv_1_t123",
                "type": "loop_iteration",
                "energy_joules": 0.0002,
                "time_seconds": 0.00005 }, // ... more checkpoints ]
         }
}
```

4 Validation and Benchmarking

We validate CodeGreen's measurement accuracy and configurable overhead using the Computer Language Benchmarks Game (CLBG) [13], which provides standardized CPU-intensive workloads across Python, C, C++, and Java. We deliberately selected short-running benchmarks because they represent the most challenging measurement scenario: smaller scripts with finer granularity amplify instrumentation overhead relative to execution time, requiring the tool to demonstrate accurate attribution even under adverse conditions.

Experimental Setup Experiments were run on an AMD Zen 4 processor, monitoring RAPL package energy. Protocol involved following the best practices from literature [20], 3 warmup iterations and 30 measured iterations per configuration, reporting mean and 95% confidence intervals. We evaluated three benchmarks across four languages: **nbody** (compute-heavy), **fannkuchredux** (CPU-bound), and **binarytrees** (memory-intensive), testing CodeGreen at both coarse (project-level) and fine (method-level) granularities.

Instrumentation Overhead and Checkpoint Granularity For coarse-grained instrumentation with only 2 checkpoints (program entry and exit), raw overhead appears high for fast-running benchmarks: C/C++ show 311–322% overhead for 40ms execution. However, as discussed in Section 3, this is dominated by the fixed 125ms library loading cost, which becomes negligible as program execution time increases. After applying overhead normalization (Sec. 3), the true measurement overhead is only 9.1–9.6% for C/C++, while Python and Java show negligible overhead as their longer runtimes (2.92s and 103ms respectively) amortize the fixed cost.

For fine-grained method-level instrumentation, overhead scales linearly with checkpoint count: **fannkuchredux** with 4 checkpoints incurs 1.5–11.3% normalized overhead, **binarytrees** with 20 checkpoints shows 7.3–22%, and **nbody** with 1 million checkpoints exhibits 607% overhead (C)—an intentional stress test demonstrating per-invocation measurement capability. This deterministic scaling enables users to tune granularity based on optimization needs: coarse-grained measurement (2 checkpoints) provides low-overhead whole-program profiling comparable to `perf`, while fine-grained instrumentation offers method-level attribution unavailable in process-level tools, accepting higher overhead for detailed hotspot identification. Unlike decorator-based profilers, whose overhead is unpredictable and compounds with execution complexity, CodeGreen's predictable, near-constant cost enables precise prediction and control of measurement impact.

Measurement Accuracy We compare CodeGreen's energy measurements against Linux `perf`, which serves as ground truth by reading directly from RAPL Model Specific Registers. CodeGreen achieves $R^2 = 0.9934$ correlation with mean absolute error of 10.9%, demonstrating high measurement fidelity. The systematic offset, where CodeGreen reports slightly higher values, arises from checkpoint instrumentation introducing additional computation and CodeGreen measuring energy between explicit checkpoints while perf captures total process energy including initialization. The strong correlation confirms that CodeGreen accurately tracks energy consumption patterns without measurement validity being compromised by instrumentation overhead.

Linearity and Scalability Energy consumption should scale linearly with computational work for constant-complexity algorithms [20]. We verify this by executing benchmarks with increasing workload sizes. CodeGreen achieves $R^2 = 0.9997$ for energy-workload linearity, confirming that the asynchronous buffer management introduces no non-linear artifacts. This near-perfect linearity validates that the lock-free circular buffer architecture maintains measurement fidelity even under extreme checkpoint densities, with no buffer contention or scaling noise as workload intensity increases.

Cross-Language Consistency We validate language extensibility by comparing energy profiles across Python, C, C++, and Java. Measurements reveal expected efficiency orderings: compiled languages consume significantly less energy than interpreted languages, with Java falling in the middle due to JIT compilation. Normalized overhead remains consistent within language families after accounting for fixed initialization and per-checkpoint costs. For example, coarse-grained instrumentation shows 9.1% overhead for C and near-zero for Python, while fine-grained instrumentation with 4 checkpoints incurs 1.5% (C) to 11.3% (C++) overhead. Despite native execution time differences spanning three orders of magnitude (40ms for C vs 2.92s for Python in **nbody**), CodeGreen provides consistent, predictable measurements across all tested languages, validating the architecture's language-agnostic design.

5 Implications

CodeGreen's design enables energy optimization workflows across multiple stakeholders. **For developers**, IDE integration would enable early-stage optimization by profiling energy during local development, highlighting hotspots through inline annotations, and preventing energy regressions before production. **For practitioners**, the platform eliminates tooling fragmentation, providing a consistent methodology across heterogeneous stacks (e.g., Python ML, C++ inference, GPUs) to replace disparate, incompatible profilers. **For researchers**, the structured JSON output facilitate LLM-driven "measure-optimize-validate" loops for AI-driven green software engineering. The deterministic overhead model ensures reproducible measurements for comparative studies, while extensibility supports emerging hardware and languages without core modifications. **For quality assurance teams**, fine-grained energy regression testing enables method-level budget assertions, pinpointing specific functions causing increased consumption rather than detecting only application-level regressions. By supporting commodity hardware with automated instrumentation, CodeGreen democratizes energy optimization, making sustainability a tractable engineering objective for startups, small teams, and educational contexts where energy efficiency can be taught alongside time complexity.

6 Threats to Validity

Construct Validity. Reliance on RAPL as ground truth imposes a 1ms resolution limit and platform-specific measurement characteristics. We mitigated this through long-running workloads and validated strong correlation with RAPL to confirm physical validity. Linear interpolation between samples introduces estimation error for sub-millisecond code blocks, though near-perfect energy-workload linearity suggests no systematic bias. **Internal Validity.** The extreme overhead in **nbody** with 1M checkpoints represents an intentional stress test demonstrating per-invocation measurement capability; standard coarse-grained usage incurs negligible overhead. The deterministic scaling validates that overhead is a configurable trade-off rather than stochastic measurement noise. **External Validity.** Experiments focus on x86 architecture, though CodeGreen's modular `EnergyProvider` interface supports extensibility to AMD, NVIDIA, and ARM platforms. Consistent measurements across Python, C, C++, and Java demonstrate robust applicability across diverse ecosystems, though validation on more architectures would strengthen generalizability claims.

7 Conclusions

CodeGreen addresses critical limitations in software energy profiling through an asynchronous producer-consumer architecture that decouples instrumentation from measurement. Achieving low overhead for whole-program profiling while enabling fine-grained method-level attribution at predictable cost. The modular design provides hardware and language extensibility supporting diverse execution environments. Validation demonstrates strong correlation with RAPL ground truth and near-perfect energy-workload

linearity across multiple languages and computational patterns. By bridging coarse-grained monitoring and manual instrumentation, CodeGreen provides developers with practical infrastructure for algorithmic energy optimization across heterogeneous computing environments.

References

[1] Marcelo Amaral, Huamin Chen, Tatsuhiro Chiba, Rina Nakazawa, Sunyanan Choochotkaew, Eun Kyung Lee, and Tamar Eilam. 2023. Kepler: A framework to calculate the energy consumption of containerized applications. In *2023 IEEE 16th international conference on cloud computing (CLOUD)*. IEEE, 69–71.

[2] Mohammed chakib Belgaid, Romain Rouvoy, and Lionel Seinturier. 2019. *Pyjoules: Python library that measures python code snippets.* https://github.com/powerapi-ng/pyJoules

[3] Max Brunsfeld, Amaan Qureshi, Andrew Hlynskyi, Will Lillis, ObserverOfTime, dundargoc, Phil Turnbull, Timothy Clem, Douglas Creager, Christian Clason, Andrew Helwer, Antonin Delpeuch, Daumantas Kavolis, Riley Bruins, Michael Davis, Ika, Amin Ya, Tuan-Anh Nguyen, bfredl, Stafford Brunk, Matt Massicotte, Niranjan Hasabnis, Rob Rix, James McCoy, Mingkai Dong, Samuel Moelius, Steven Kalt, Josh Vera, and Kolja. 2025. *tree-sitter/tree-sitter: v0.26.3.* doi:10.5281/zenodo.17921700

[4] Semen Andreevich Budennyy, Vladimir Dmitrievich Lazarev, Nikita Nikolaevich Zakharenko, Aleksei N Korovin, OA Plosskaya, Denis Valer'evich Dimitrov, VS Akhripkin, IV Pavlov, Ivan Valer'evich Oseledets, Ivan Segundovich Barsola, et al. 2022. Eco2ai: carbon emissions tracking of machine learning models as the first step towards sustainable ai. In *Doklady mathematics*, Vol. 106. Springer, S118–S128.

[5] Stefano Corda, Bram Veenboer, and Emma Tolley. 2022. Pmt: Power measurement toolkit. In *2022 IEEE/ACM International Workshop on HPC User Support Tools (HUST)*. IEEE, 44–47.

[6] Benoit Courty, Victor Schmidt, Goyal-Kamal, MarionCoutarel, Boris Feld, Jérémy Lecourt, LiamConnell, SabAmine, inimaz, supatomic, Mathilde Léval, Luis Blanche, Alexis Cruveiller, ouminasara, Franklin Zhao, Aditya Joshi, Alexis Bogroff, Amine Saboni, Hugues de Lavoreille, Niko Laskaris, Edoardo Abati, Douglas Blank, Ziyao Wang, Armin Catovic, alencon, Michał Stęchły, Christian Bauer, Lucas-Otavio, JPW, and MinervaBooks. 2024. *mlco2/codecarbon: v2.4.1.* doi:10.5281/zenodo.11171501

[7] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268. doi:10.1109/JSSC.1974.1050511

[8] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM computing surveys (CSUR)* 35, 2 (2003), 114–131.

[9] Guillaume Fieni, Daniel Romero Acero, Pierre Rust, and Romain Rouvoy. 2024. PowerAPI: A Python framework for building software-defined power meters. *Journal of Open Source Software* 9, 98 (June 2024), 6670. doi:10.21105/joss.06670

[10] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. 2020. Smartwatts: Self-calibrating software-defined power meter for containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 479–488.

[11] Raphael Fischer. 2025. Ground-Truthing AI Energy Consumption: Validating CodeCarbon Against External Measurements. arXiv:2509.22092 [cs.AI] https://arxiv.org/abs/2509.22092

[12] Linux Foundation. 2017. How to build a basic cyclic application. https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/cyclic. [Accessed 27-01-2026].

[13] Isaac Gouy. [n. d.]. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/

[14] Hongyu Hè, Michal Friedman, and Theodoros Rekatsinas. 2023. EnergAt: Fine-Grained Energy Attribution for Multi-Tenancy. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems (HotCarbon '23)*. ACM, 1–8. doi:10.1145/3604930.3605716

[15] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. 2018. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3, 2 (2018), 1–26.

[16] Adel Noureddine. 2022. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *18th International Conference on Intelligent Environments (IE2022)*. Biarritz, France.

[17] Benoit Petit. 2023. *scaphandre.* https://github.com/hubblo-org/scaphandre

[18] Saurabhsingh Rajput. 2026. *CodeGreen: A Modular Energy Measurement System for Multi-Language Software.* https://smart-dal.github.io/codegreen/

[19] Saurabhsingh Rajput. 2026. *Demo CodeGreen.* https://smart-dal.github.io/codegreen/demo

[20] Saurabhsingh Rajput, Tim Widmayer, Ziyuan Shang, Maria Kechagia, Federica Sarro, and Tushar Sharma. 2024. Enhancing energy-awareness in deep learning through fine-grained energy measurement. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–34.

[21] C.E. Shannon. 1949. Communication in the Presence of Noise. *Proceedings of the IRE* 37, 1 (1949), 10–21. doi:10.1109/JRPROC.1949.232969

[22] Saurabh Singh. 2026. *SMART-Dal/codegreen: v0.1.2.* doi:10.5281/zenodo.18371772

[23] Ptidej Team. 2023. *JoularJX - Learn About Your Java App's Power Consumption.* https://blog.ptidej.net/joularjx/ Accessed: 2026-01-21.

[24] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th international conference on parallel processing workshops*. IEEE, 207–216.

[25] Steven van der Vlugt, Leon Oostrum, Gijs Schoonderbeek, Ben van Werkhoven, Bram Veenboer, Krijn Doekemeijer, and John W. Romein. 2025. PowerSensor3: A Fast and Accurate Open Source Power Measurement Tool. arXiv:2504.17883 [cs.PF] https://arxiv.org/abs/2504.17883

[26] Roberto Verdecchia, Patricia Lago, Christof Ebert, and Carol De Vries. 2021. Green IT and green software. *IEEE software* 38, 6 (2021), 7–15.

[27] Vincent M Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. 2012. Measuring energy and power with PAPI. In *2012 41st international conference on parallel processing workshops*. IEEE, 262–268.

[28] Wikipedia contributors. 2025. Aliasing — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Aliasing&oldid=1319662380. [Online; accessed 27-January-2026].

[29] Wikipedia contributors. 2026. Network Time Protocol — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Network_Time_Protocol&oldid=1334767421. [Online; accessed 27-January-2026].

[30] Zeyu Yang, Karel Adamek, and Wesley Armour. 2024. Accurate and Convenient Energy Measurements for GPUs: A Detailed Study of NVIDIA GPU's Built-In Power Sensor. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–17. doi:10.1109/sc41406.2024.00028

[31] Josh You. 2025. How much energy does ChatGPT use? https://epoch.ai/gradient-updates/how-much-energy-does-chatgpt-use Accessed: 2026-01-26.

A Appendix: Tool Demonstration Walkthrough

This appendix provides a step-by-step walkthrough of the CodeGreen command-line interface (CLI), demonstrating core capabilities from installation to fine-grained energy measurement. We present a practical usage guide with screenshots showing each stage of a typical workflow.

A.1 Availability

CodeGreen is an active, open-source project available through the following resources:

- **Website & Documentation:** https://smart-dal.github.io/codegreen/
- **Source Code:** https://github.com/SMART-Dal/codegreen
- **Video Demonstration:** https://smart-dal.github.io/codegreen/demo/
- **License:** Mozilla Public License 2.0 (MPL 2.0)

The project website (Fig. 2) provides comprehensive documentation, video demonstrations, installation guides, and API references to support users from initial setup through advanced usage scenarios.

A.2 Quick Start Guide

Table 1 provides a navigation index for the demonstration figures.

Step 1: Installation. Clone the repo and run installation script:

```
git clone https://github.com/SMART-Dal/codegreen.git
cd codegreen
./install.sh
```

The installer automatically detects your system architecture, compiles the Native Energy Measurement Backend (NEMB), and installs Python bindings (Fig. 3).

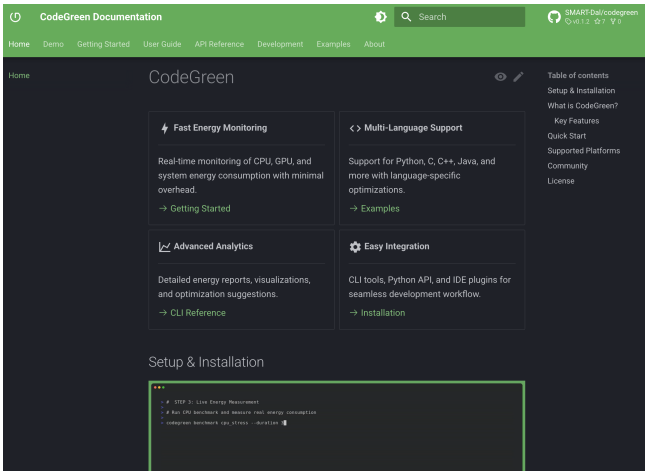

Figure 2: CodeGreen Website: The official website at https://smart-dal.github.io/codegreen/ provides installation guides, API references, usage examples, and demo videos.

Table 1: Navigation Index for CodeGreen Demonstration

Fig.	Stage	Purpose
3	Install	Setup & Compilation
4	Help	Explore subcommands
5	Init	Detect hardware sensors
6	Info	Verify system config
7	Bench	Live stress testing
8	Analyze	Static AST instrumentation
9	Measure	Fine-grained profiling
10	JSON	CI/CD data export
11	Config	Tuning granularities
12	Doctor	System diagnostics

Available Commands: After installation, explore the available CLI commands to familiarize yourself with CodeGreen's capabilities (Fig. 4):

```
codegreen --help
```

The help interface organizes commands into three main categories: measurement operations for profiling energy consumption, analysis tools for static code inspection, and system management utilities for configuration and diagnostics.

Step 2: Initialize Hardware Sensors. Configure CodeGreen to detect and access available energy sensors:

```
codegreen init-sensors
```

This command identifies Intel RAPL, NVIDIA NVML, and AMD ROCm interfaces, configuring non-root access permissions where possible (Fig. 5).

Verify successful initialization:

```
codegreen info
```

The output confirms available sensors, RAPL domains (Package, Core, DRAM), and GPU devices (Fig. 6).

Step 3: Validate with Built-in Benchmark. Test the measurement pipeline with a CPU stress workload:

```
codegreen benchmark --duration 10
```

This reports total energy consumption (Joules), average power (Watts), and execution time with confidence intervals (Fig. 7).

Step 4: Analyze Source Code. Before instrumenting, preview what will be measured using static analysis:

```
codegreen analyze script.py
```

Based on user-specified granularity in the configuration (e.g., functions, loops, or classes), Tree-sitter AST queries [3] identify and target the corresponding code constructs for instrumentation without executing the program (Fig. 8). This automated analysis provides developers with granular control over measurement scope while eliminating manual checkpoint placement.

Step 5: Measure Energy Consumption. Execute fine-grained profiling with automatic instrumentation:

```
codegreen measure script.py
```

CodeGreen instruments the code, runs it, and attributes energy to specific functions, identifying hotspots (Fig. 9).
For CI/CD integration, export results as JSON:

```
codegreen measure script.py --output json
```

This structured format enables automated regression testing (Fig. 10).

Step 6: Configure Measurement Granularity. Tune profiling behavior to balance precision and overhead:

```
codegreen config
```

Adjust sensor polling intervals, accuracy thresholds, and instrumentation granularity (Fig. 11).

Step 7: System Diagnostics. Verify installation integrity and sensor health:

```
codegreen doctor
```

This checks file permissions, validates dependencies, and tests sensor read operations (Fig. 12).

A.3 Typical Workflow Summary

A typical energy optimization workflow is as follows:

(1) **Initialize:** Detect hardware sensors with `init-sensors` and verify configuration using `info`
(2) **Baseline:** Establish baseline energy consumption using `benchmark` to validate measurement pipeline
(3) **Profile:** Identify energy hotspots by running `measure script` with appropriate granularity settings
(4) **Optimize:** Refactor hotspots based on profiling results
(5) **Validate:** Re-measure optimized code to quantify energy improvements and prevent regressions
(6) **Automate:** Integrate JSON output into CI/CD pipelines for continuous energy regression testing

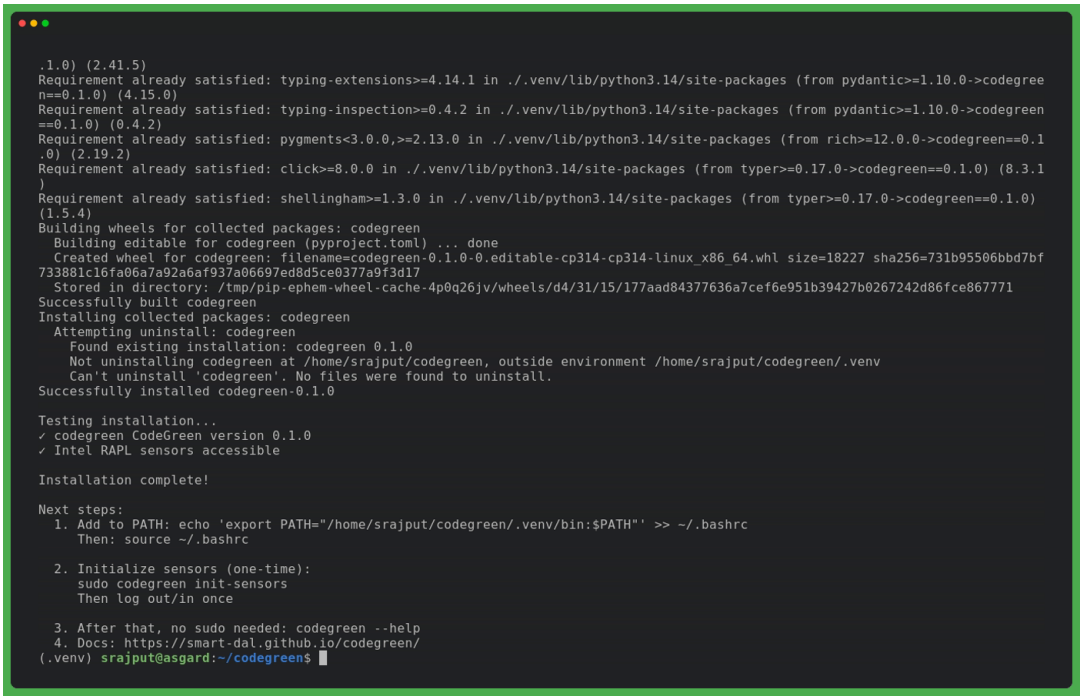

Figure 3: Installation: Running `./install.sh` automatically detects the host architecture, compiles NEMB, and installs Python bindings and CLI tools.

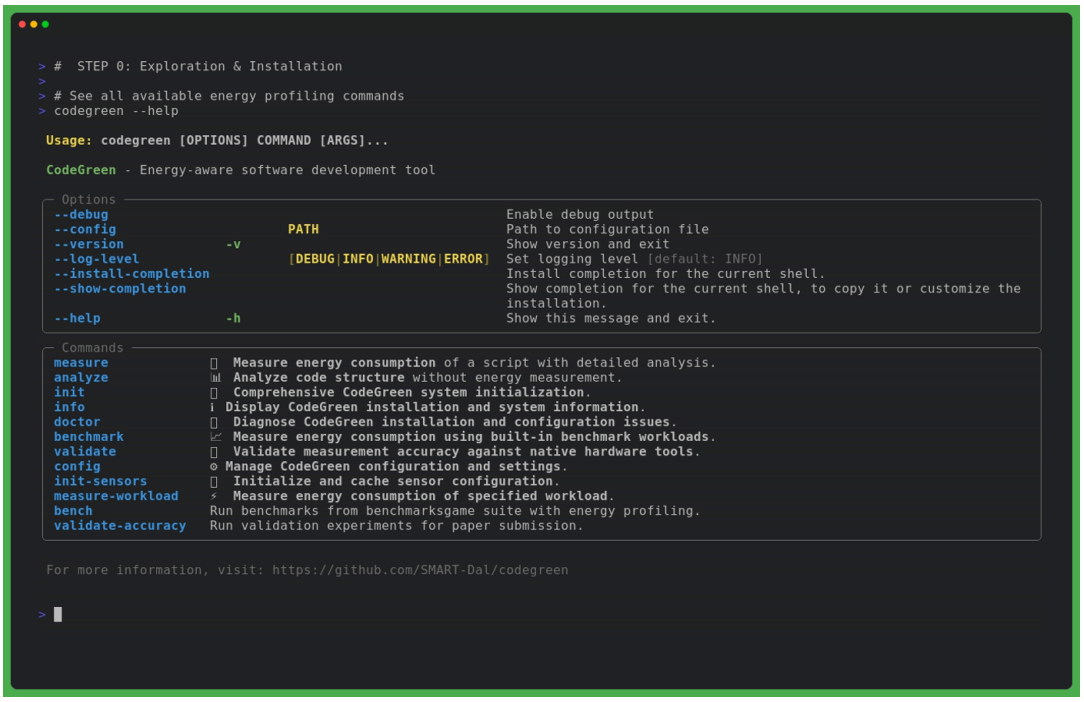

Figure 4: Command Overview: The `codegreen –help` command displays the complete CLI interface, categorizing subcommands into measurement (`measure`, `benchmark`), analysis (`analyze`), and system management (`init-sensors`, `config`, `doctor`, `info`) tasks.

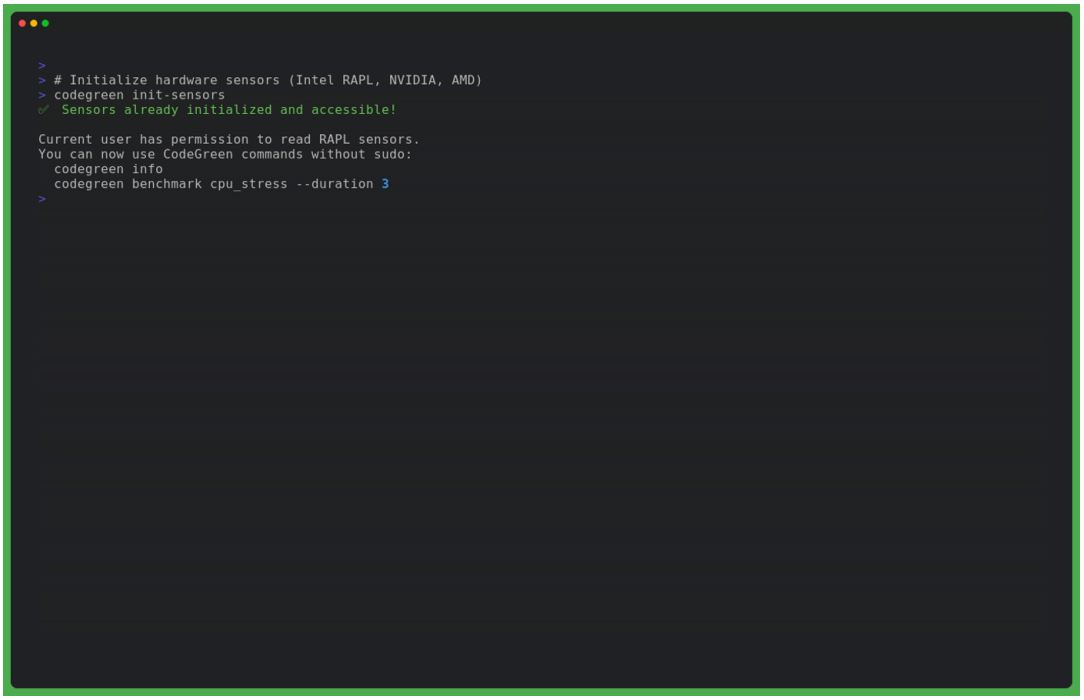

Figure 5: Sensor Initialization: The `codegreen init-sensors` command detects Intel RAPL, NVIDIA NVML, and AMD ROCm interfaces, configuring non-root access permissions.

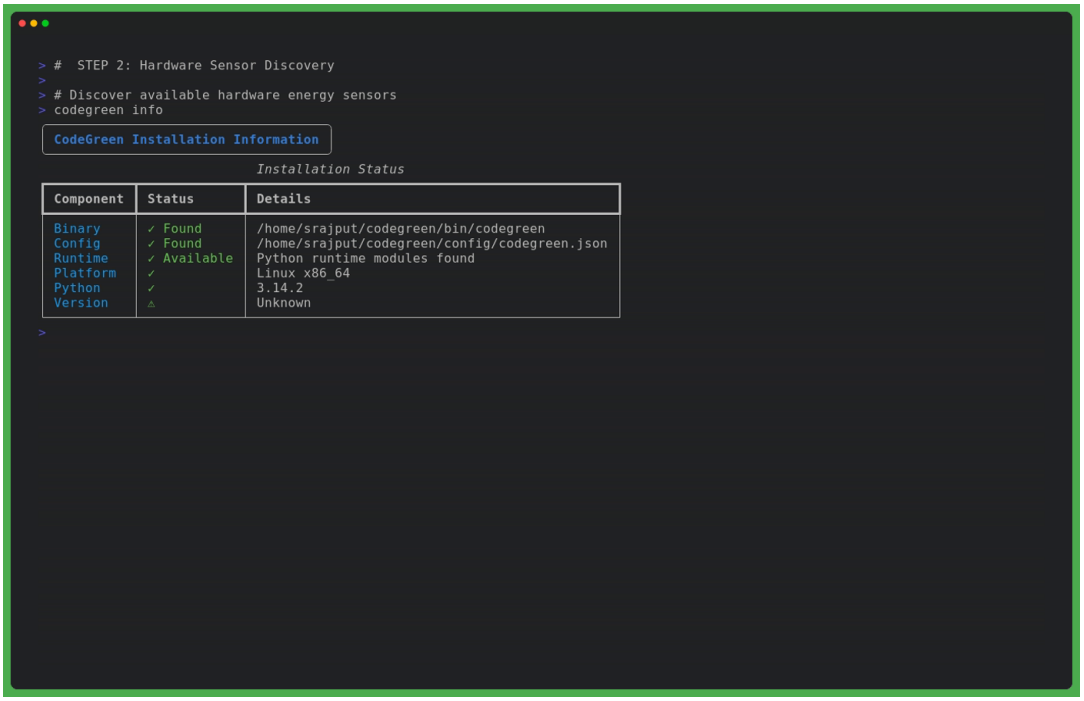

Figure 6: System Verification: The `codegreen info` command reports active configuration, confirming available sensors.

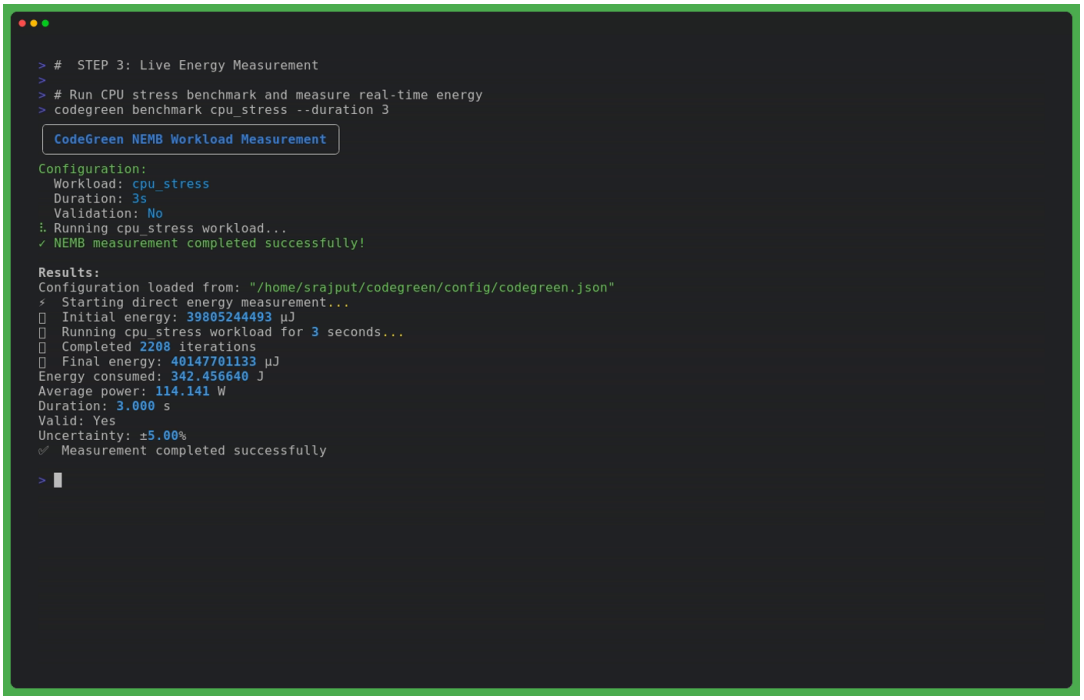

Figure 7: Live Benchmarking: The `codegreen benchmark` command validates the measurement pipeline with a CPU stress test.

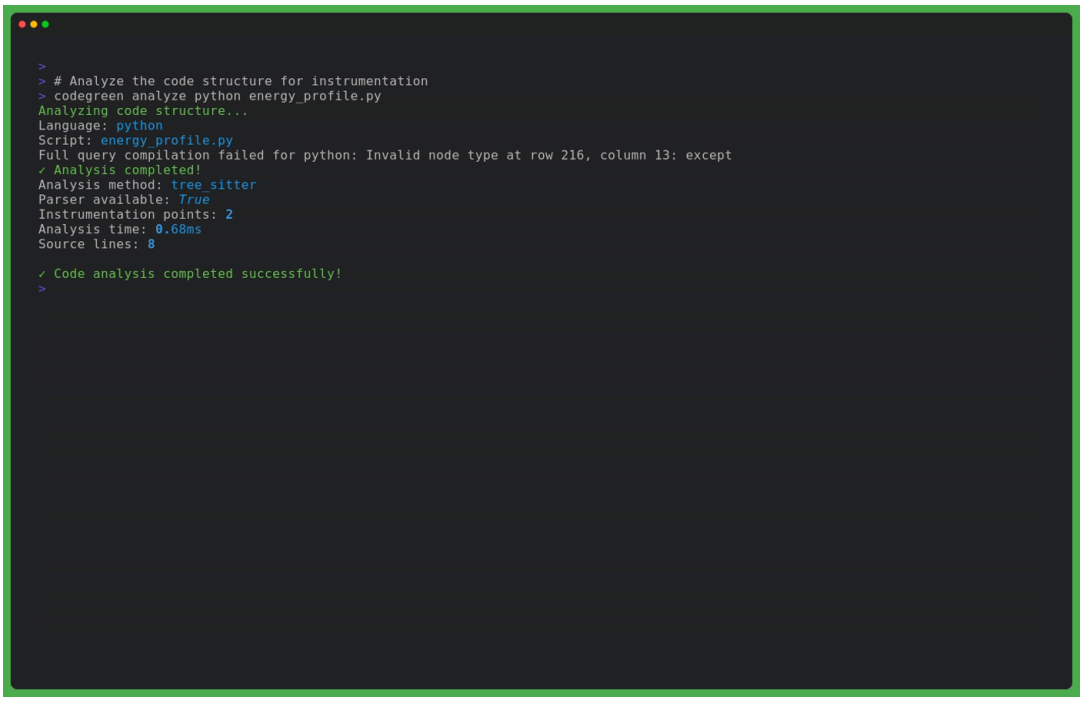

Figure 8: Static Analysis: The `codegreen analyze script.py` command uses Tree-sitter AST queries to identify instrumentation points (functions, loops, classes) based on user-configured granularity settings, without executing code.

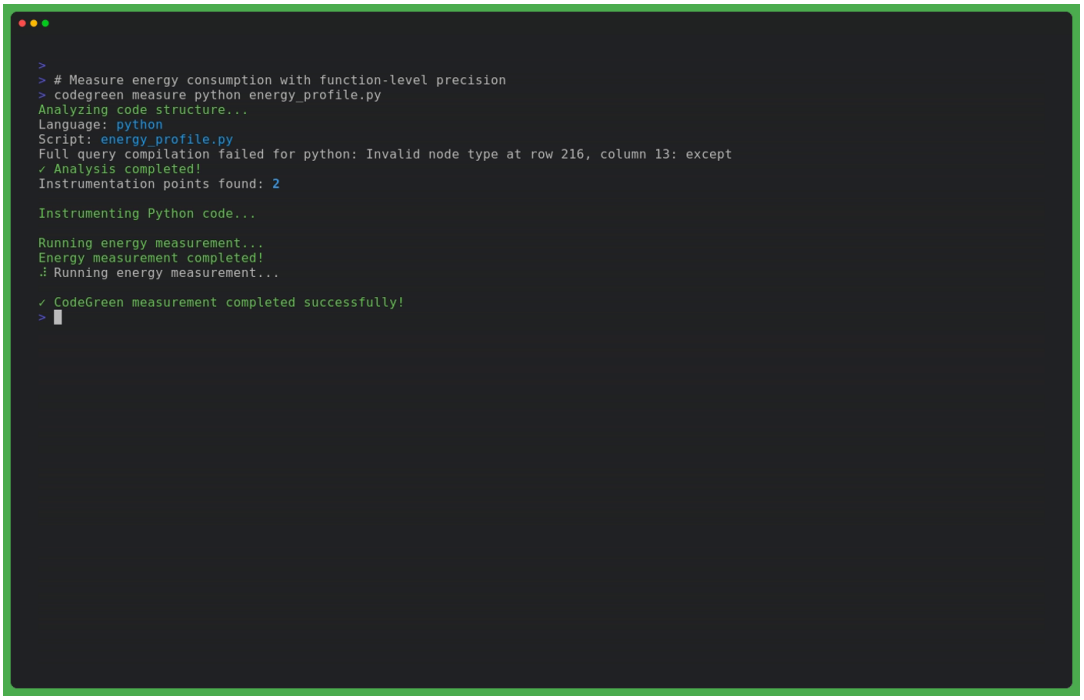

Figure 9: Fine-Grained Measurement: The `codegreen measure` command instruments code and attributes energy to specific functions.

Figure 10: JSON Export: Using `–output json` enables CI/CD integration for automated energy regression testing.

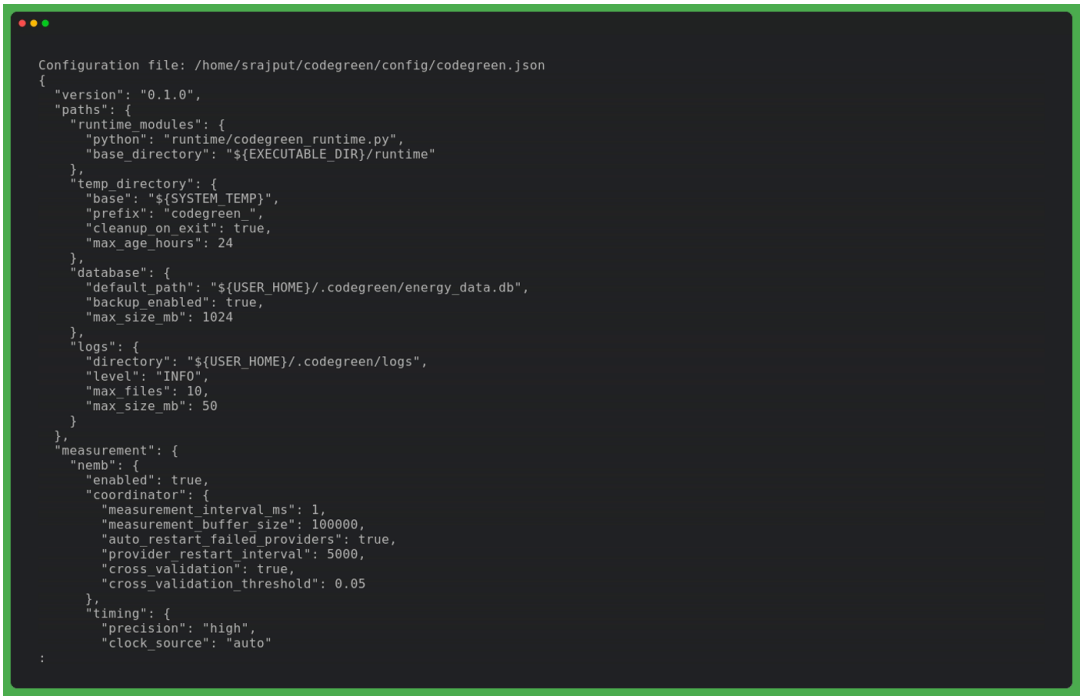

Figure 11: Configuration: The `codegreen config` command tunes sensor polling intervals and instrumentation granularity.

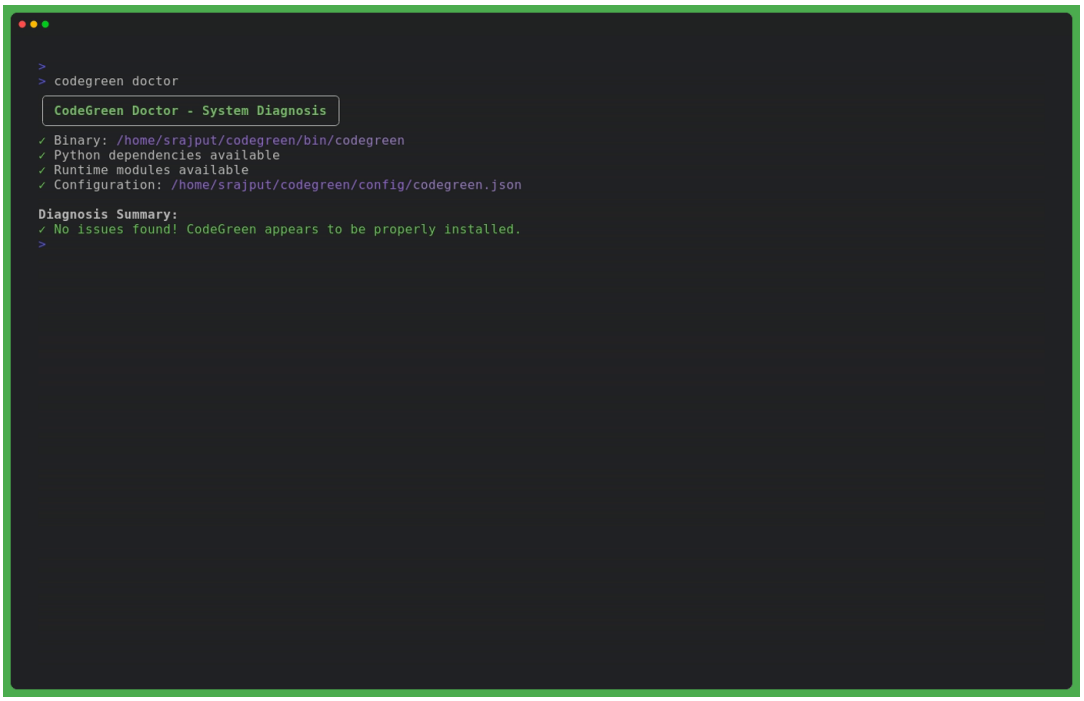

Figure 12: Diagnostics: The `codegreen doctor` command verifies file permissions, validates dependencies, and tests sensor health.