CS4575: Sustainable SE - Project Energy Profiling of Static Analysis Tools

Andrea Onofrei
TU Delft
Delft, Netherlands
aonofrei@tudelft.nl

Ayush Kuruvilla
TU Delft
Delft, Netherlands
akuruvilla@tudelft.nl

Sahar Marossi
TU Delft
Delft, Netherlands
@student.tudelft.nl

Yulin Chen
TU Delft
Delft, Netherlands
yulinchen@tudelft.nl

Abstract

With increasing interest in software security, more automated tools for vulnerability detection and secure coding standard enforcement, also known as static analysis tools, are being utilized. On the other hand, with sustainability fast becoming a critical non-functionality requirement, effectiveness is no longer the only parameter that needs to be considered. This paper outlines a published pypi framework which captures and compares the energy usage of static analysis tools used in security audit definitions, namely Bandit and Semgrep. The goal of this research is to empower developers and enable them to make educated decisions regarding energy usage by examining the impact of various changes in configurations and rule sets toward the energy expenditure during the process.

CCS Concepts

• **Static analysis tools → Sustainable SE**.

Keywords

Energy Profiling, Static analysis tools

ACM Reference Format:
Andrea Onofrei, Ayush Kuruvilla, Sahar Marossi, and Yulin Chen. 2025. CS4575: Sustainable SE - Project Energy Profiling of Static Analysis Tools . In *CS4575-Q3-25: Sustainable SE, TU Delft*. ACM, New York, NY, USA, 12 pages.

1 Introduction

Software Engineering (SE) is a constantly developing field, and software applications are becoming increasingly complex by the day [9]. Academic studies show that there is a direct correlation between software complexity and security vulnerabilities [1]. This introduces additional demands within all phases of the software development life cycle (SDLC), especially within the context of software security practices.

CS4575-Q3-25, TU Delft

Security concerns are considered to be an integral part of the SDLC [14]. The consequences of inadequate security within a software system could potentially be detrimental, especially due to the increasing reliance on software systems in various sensitive domains [11]. For example, the banking and financing industry deals with confidential data and large transactions which may be prone to security breaches. A study by Shehab et al. (2024) derives that the average cost of a data breach ranges within the order of millions of dollars [13].

A highly important software security practice is **code reviewing for security** [10]. This can be achieved through **static-analysis tools**, which are tools that allow for the identification of many common coding problems through automated means before a program is released [2]. This includes examining code for bugs, vulnerabilities, and other quality issues without executing the source code. Tools like Bandit and Semgrep are common tools used within the security domain to detect insecure coding practices in order to ensure better coding practices at scale.

While static analysis tools help improve code quality, and in turn, security, they also have the potential to consume computing resources. With the increasing need of secure code due to ubiquitous software systems, static analysis is more frequently integrated into CI/CD pipelines, often without consideration for its environmental cost. This could lead to significant and unnecessary energy usage, especially in larger code bases. While automated quality assurance processes are crucial for software reliability, their electricity consumption raises environmental concerns.[15]

Sustainable software engineering is an important goal to follow, to develop software systems that minimize environmental impact via energy optimization. As security-related static analysis tools become increasingly important, so does the need to assess their sustainability. This project aims to bridge that gap by profiling energy consumption across different tools and configurations, investigating the potential trade-off between differing configurations across different Open-source software (OSS) projects, and enabling developers to optimize their tool usage both from a security and sustainability standpoint.

The research objectives are the following:

- Investigate energy consumption patterns in security-focused static analysis tools.

- How do rule set complexity and configuration choices affect the energy efficiency of static analysis tools in different codebase contexts?
- Can energy usage data be effectively used to guide configuration selection for secure static analysis without compromising detection accuracy?.

To tackle these objectives, this report documents the We propose an energy profiling framework to:

- Measure energy usage during static analysis tool runs.
- Compare different rule sets and configurations within and across tools.
- Generate actionable reports to help developers reduce energy consumption.

2 Background and Related Work

Although efforts to quantify and mitigate energy dissipation attributed to software date back over a decade, the initial efforts were primarily directed towards the runtime efficiency of software systems and not much development tooling. One original approach was Hindle's Green Mining (2015) [7] which attempted to relate software revisions to power measurements with the hope of discovering how code modifications impact energy utilization. Such efforts set the stage to begin considering energy as a quantitative variable in software engineering research.

Later, there was energy conscientious software development that came into existence. Pang et al.'s work (2016) [12] reported on a survey given to programmers and noted an overall ignorance of the issues surrounding software energy dissipation and stressed the importance of education and tools in this domain. In response to this gap in awareness, researchers like Chowdhury et al. (2019) developed techniques to give developers feedback on energy usage. Chowdhury's GreenScaler project [3], for instance, trained models to estimate energy consumption of code by using automatically generated tests. The underlying idea is that if developers are informed about which tests or code paths are energy-hungry, they can make more sustainable design choices.

Another venture of green software research has been to create tools and methodologies for energy profiling. For example, Di Nucci et al. (2017) introduced PETrA [5], a software-based tool to estimate the energy profile of Android applications. Although PETrA targets mobile apps, its existence reflects a growing toolkit for software energy analysis in general.

Recently, attention has turned toward the energy overhead of development infrastructure and processes.As described by Verdecchia et al. (2021) [6], automation of tools and developer workflows are part of the equation for greener IT. Zaidman (2024) [15] reported that the automated building and testing of software projects consumes a significant portion of the projects' energy. In his case study, he found that the figures were remarkably different between the projects. Some of them, made only a couple of watt-hours during a build, whereas, others consumed energy over tens of kWh. Cruz and Abreu (2020) [4] also discovered that some mobile testing automation frameworks are more energy efficient than others, disproving the common assumption. Collectively, these works suggest that not all tools or practices are equal with respect to energy – the selection of frameworks, frequency of tests, and infrastructure used all affect electricity consumption to some extent.

Although green software engineering literature has initiated the investigation of other tools such as testing, build processes, and even runtime diagnostics, there is still a lack of coverage on the topic of security focused static analysis tools. Most existing research either treats static analysis generically as part of overall CI energy costs or focuses on how static analysis can help reduce software energy use – not on the energy cost of running the analyzers themselves. For example, Zaidman's study mentions static analysis as one of several automated quality practices but concentrates on testing and integration. Likewise, studies such as Hindle's Green Mining or Cruz and Abreu's work on test frameworks did not examine static code scanners.

One recent study by Brosch [8] considered the influence of static code analysis (Pylint) on a game systems algorithm's energy consumption, finding that certain static checks (e.g., for inefficient code) could lead to more energy-efficient codebases – but this was more type checking as compared to security static analysis testing.

To summarize, tools such as Bandit and Semgrep have not had their energy profiles estimated to date. It is likely that regularly executed security-focused static analyses (which are sometimes done for every build or pull request) accumulate over time and can draw power, especially if they are done on bigger codebases. This gap is filled by our work which provides the energy profile of Bandit and Semgrep. We focus on understanding how such security tools impact the environment and whether their design is sufficiently optimized for "green IT" by measuring energy consumption and duration of execution in real-life settings.

In this paper, we add to the emerging literature that sits at the boundary of software energy analysis and DevSecOps. This study illustrates the future of sustainable software engineering, in which decisions made, such as the selection of security scanning tools, consider effectiveness alongside sustainability. Gaining insights into the cost of energy for automated security checks will ultimately enable practitioners to mitigate the impacts of code security on environmentally friendly software development practices, thereby simultaneously improving quality and sustainability.

3 Methodology

3.1 Objective of the Study

In this research, we study the energy efficiency of static application security testing (SAST) tools in relation to Python based codebases. We investigate the functionalities and security aspects of two popular SAST tools, `Bandit` and `Semgrep`, under varying configurations. We also quantify the energy consumed by each scan using a Python tool developed by us, called `sast-energy-monitor`, which incorporates `energibridge` for detailed energy consumption measurement.

3.2 Selection of Open Source Projects

To evaluate the energy consumption of static analysis tools in real-world scenarios, we selected a set of open-source Python projects. These projects vary in size, complexity, and popularity to ensure generalizable results.

Project Repository 1: Deepseek

- **Language:** Python
- **Lines of Python Code:** 1,202
- **Purpose:** Lightweight AI tools and utilities.
- **Popularity:** Moderate (growing adoption in ML communities)

Rationale for Selection:

- Compact codebase suitable for low-overhead static analysis.
- Includes AI-specific logic and processing utilities.
- Serves as a baseline for energy usage in smaller repositories.

Project Repository 2: Requests

- **Language:** Python
- **Lines of Python Code:** 9,164
- **Purpose:** A simple, yet elegant HTTP library for Python
- **Popularity:** High (well-maintained, widely used in the Python ecosystem)

Rationale for Selection:

- Popular and mature Python library used in thousands of projects.
- Contains both application logic and input/output handling, offering a rich target for static security analysis.
- Its high usage in production environments makes it a practical candidate for sustainability evaluation.

Project Repository 3: vLLM

- **Language:** Python
- **Lines of Python Code:** 286,537
- **Purpose:** High-throughput and memory-efficient LLM serving engine
- **Popularity:** High (widely adopted in large-scale LLM serving applications)

Rationale for Selection:

- Large codebase introduces challenges for static analysis tools.
- Includes multithreading, CUDA integration, and complex logic layers.
- Ideal for stress-testing the tools' scalability and energy profiles.

Security Static Analysis Tools

`Bandit` is a static analysis tool specialized in Python and scans source files by traversing their AST to discover known security vulnerabilities. In the case of security issues, a specific rule containing a code such as `B101` (which corresponds to a misuse of `assert`) is predefined. `Semgrep`, on the other hand, is a powerful structural pattern matcher that combines knowledge of programming languages and scans the source code to extract structural code patterns. Its rule definitions permit generic context and context-sensitive checks thus providing a powerful customizable vulnerability scanner.

For the purpose of a comprehensive evaluation, we employed both tools in two configurations, termed *loose* and *strict*. They both represent different degrees of security checks realized through custom configuration files.

Configuration Strategy

The energy monitoring tool, `sast-energy-monitor`, allows for either a strict or loose configuration level and calls the appropriate scanner. For `Bandit`, the loose mode refers to a subset configuration (.bandit-basic) which only performs two simple checks: scanning for assert statements (B101) and dynamic exec usage (B102). This is a lightweight scan that is useful for rapid iteration.

`Bandit strict` mode uses the full `.bandit` configuration file which contains more than 60 rules. These rules include scanning for assert statements and dangerous imports such as `pickle` and `subprocess`, weak cryptographic methods like MD5 with low-entropy keys, hardcoded secrets, XML parser bombs, insecure network connections, and others. This configuration follows industry standards by incorporating many CWE and OWASP rules that enable to perform production-level scans.

With `Semgrep`, the loose configuration has a simplistic YAML rule set targeting hardcoded JWT secret tokens, which in Python applications, represent a common but sensitive vulnerability. The strict mode uses the complete Semgrep Registry ruleset under the `p/bandit` policy with 90 rules.

Energy Monitoring Tool

To allow for reusability of our code, we created the Python package `sast-energy-monitor`, which encapsulates the invocation of Bandit or Semgrep scans, with varying configurations using `energibridge` in one package. `Energibridge` in turn was built by leveraging low level tools such as Intel RAPL to access energy usage metrics at runtime.

By automating both the scanning and measurement processes, this methodology ensures consistency across experiments and allows for comparative evaluation of different scanners and configurations in terms of their energy footprint and vulnerability detection depth.

3.3 Experiment Setup

To ensure reliable and reproducible measurements of both energy consumption and security findings, each experiment followed a controlled execution run. At the start of each run, a Fibonacci warmup function was executed. This step served to stabilize CPU performance and avoid energy spikes that might occur at the onset of computation, ensuring the baseline energy draw remained consistent across all iterations.

Each test condition defined by a specific combination of the static analysis tool and its corresponding rule set, was executed thirty times. The order of these conditions was randomized before each round of testing to mitigate systematic bias from thermal or power-related fluctuations. The library automates the execution of every scan through subprocess calls while working together with Energibridge for measuring and registering total energy spent during the scan time. The results from each iteration was captured in CSV files for post analysis of tool results and energy measurements.

To avoid drift, a twenty second resting time was implemented between iterations. This break assisted in normalizing system usage to minimize the chances of performance throttling and getting more precise energy consumption measures.

All experiments were conducted in 'zen mode'. This involved disabling all non-essential background apps and services, removing external hardware peripherals, and setting screen brightness to a particular value. These steps helped ensure that every iteration captured the true nature of the scanning tools and the configurations instead of being impacted by other external noise or factors.

4 Results and Analysis

Tool Configuration: Ruleset Size and Complexity

Both Bandit and Semgrep were configured to scan for a comparable set of Python security vulnerabilities. Although the rulesets covered equivalent issues, the way each tool handles rule execution differs significantly and directly influences their energy behavior.

- **Bandit:** Bandit's rules are implemented as Python code that inspects the abstract syntax tree (AST) of each file. Each rule is essentially a function that runs over relevant AST nodes (for example, a rule might trigger when it sees an `eval()` call or a hard-coded password in the code). Bandit runs in a single thread and iterates through each file's AST, applying all relevant rules. More rules mean more checks per file, increasing runtime roughly linearly. However, Bandit's checks are generally fairly simple (looking for specific function names, constants, or patterns), which keeps each check lightweight.
- **Semgrep:** Semgrep doesn't have built-in rules in the same way; it uses an external ruleset. In this experiment, Semgrep was supplied with a ruleset equivalent to Bandit's coverage (the Semgrep community provides rulesets like p/bandit that cover the same issues as Bandit). Thus, Semgrep was effectively looking for the same kinds of security patterns. Under the hood, Semgrep compiles these rules (written in a declarative pattern syntax) into its scanning engine. Many Semgrep rules correspond one-to-one with Bandit checks, though some may be implemented with multiple pattern clauses or regexes to cover variations . Semgrep's engine will parse each source file and then apply all the pattern rules to that file's parse tree. It can do this efficiently in C/OCaml, but it does incur some overhead to load and compile the rules (especially since the ruleset was fairly large, 90 rules). By default, Semgrep also utilizes multiple threads to process files in parallel.

  Complexity Impact: In this comparison, both tools were configured with a comparable number of rules (covering similar checks), so the rule count was kept roughly constant between them. But, the nature of those rules can affect performance. Bandit's Python-based rules might short-circuit quickly if a pattern isn't found (for example, if a file has no import statements related to security, many Bandit rules do nothing on that file). Semgrep's pattern matching might involve more work upfront (it may attempt matches even if ultimately none are found, which is some overhead). In a few cases, Semgrep's rules could be more complex – for instance, a Bandit rule that checks for use of a weak hash function might simply look for the string "md5" in the code, whereas a Semgrep rule might be written to pattern-match function calls using hashlib.md5 or openssl.md5 etc., covering more variants. This can make Semgrep's analysis a bit heavier, but also more comprehensive.

Project Codebase Characteristics and Complexity:

- **DeepSeek-V3:** The codebase is small and self-contained, with no heavy concurrency or large data processing. This low complexity meant static analysis had a light workload. Bandit and Semgrep only needed to parse and check a handful of files, so the runtime was very short. In such a scenario, the fixed overhead of the tool dominates—i.e., the time/energy the tool spends initializing and loading rules can outweigh the actual scanning of code. This is why Semgrep, which has a larger upfront cost, used more energy here, whereas Bandit's simpler run had an advantage.
- **Requests:** Requests has a well-structured codebase with multiple modules (e.g., sessions, auth, adapters) but remains relatively straightforward. It primarily uses synchronous I/O and does not spawn threads internally. The moderate size means a few dozen source files. Static analysis had to process more code than DeepSeek-V3, but still at a manageable scale. Bandit's sequential scan handled this efficiently, and its overhead remained low. Semgrep had more work to do than in DeepSeek, but still needed to expend effort setting up its engine. The result was that Bandit maintained a lower total CPU energy usage on Requests as well. The project's architecture (being a typical Python library) did not present special challenges to either tool—the main factor was just code volume, where 9k LOC is still small enough that Bandit's lack of parallelism wasn't a problem.
- **vLLM:** This project is vastly more complex. It includes advanced features like request batching, asynchronous scheduling, and integration with GPU acceleration (optimized CUDA kernels are part of its design). The codebase spans hundreds of Python files and incorporates multithreading or asynchronous code to manage inference tasks. This sheer size and complexity posed a stress test for the static analysis tools. Scanning vLLM means parsing a huge number of files and checking many possible code patterns. Bandit, being single-threaded, had to process files one by one for the entire 286k LOC, which took a long time. Semgrep, on the other hand, could distribute the workload across CPU cores. The concurrency and architectural complexity of vLLM (while relevant at runtime) mostly affects the static analyzers by way of code quantity and possibly some complicated syntax. For instance, vLLM uses dynamic constructs or long chains of calls that each tool's rules need to examine. This increases the workload linearly with LOC (each additional function or class is another thing to check for vulnerabilities). The end result was that vLLM demanded far more processing, and here the ability to parallelize gave Semgrep a big edge in performance and energy. Bandit's pure Python implementation may also struggle with memory usage on such a large project (large ASTs, many objects), potentially causing slowdowns (e.g., garbage collection pauses) that add to its energy consumption.

Influence on energy use: In summary, small, simple projects (like DeepSeek-V3) don't fully utilize modern CPUs—the static analysis finishes so quickly that a tool with more overhead (Semgrep) doesn't get to amortize that cost, resulting in proportionally higher energy per LOC. Large, complex projects (like vLLM) keep the analyzer busy for much longer and can take advantage of optimizations like parallel threads. Thus, project size is a major factor: Bandit excels on smaller codebases where its lightweight, single-process approach means lower overhead, whereas on large codebases the lack of parallelism becomes a disadvantage. Meanwhile, Semgrep's design incurs a startup cost that is paid off when there's a lot of code to scan; its architecture works well in large, complex projects by leveraging concurrency and efficient parsing to handle scale.

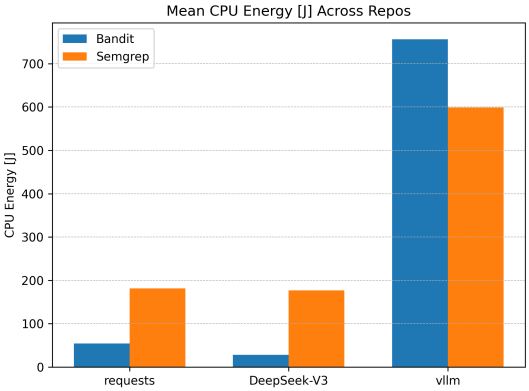


Figure 1: Energy difference between repositories

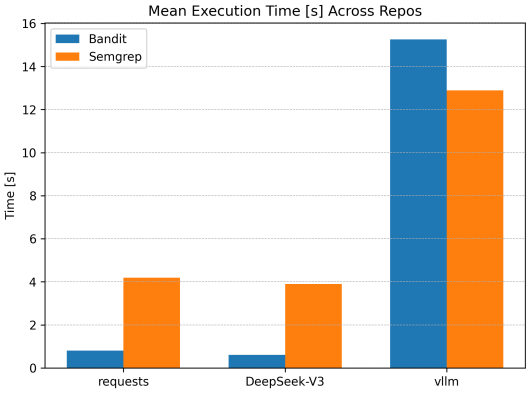


Figure 2: Time difference between repositories

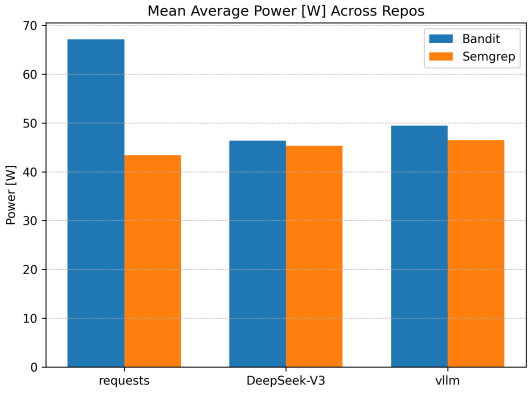


Figure 3: Power difference between repositories

CPU Power, Energy, Time, and Variability

Using EnergiBridge measurements, we compared CPU energy usage, power, and execution time for Bandit and Semgrep across three Python projects. Each tool was run 30 times to assess consistency

- **DeepSeek-V3:** As we can see in Figure 3, Bandit used 47W on average and finished in 1s. Semgrep drew similar power (45W) but took 4s to complete. This longer runtime led Semgrep to consume significantly more energy. Despite near-identical power profiles, Bandit's quick execution ensured lower energy use.
- **Requests:** Bandit completed scans in under a second, drawing an average of 65–70W, while Semgrep, as we can see in Figure 2, took 4–5 seconds at 43W. Despite lower power, Semgrep consumed 3–4× more total CPU energy due to longer runtime.
- **vLLM :** Bandit consumed more energy and took longer than Semgrep. Here, Semgrep's multithreaded execution paid off, completing faster (13s vs. 15s) and using less energy (600J vs. 750J). Power draw was nearly the same (49–50W), but the shorter runtime made Semgrep more efficient on this large project.

Key Insights and Energy Implications

- Bandit is more efficient for small and medium projects due to fast, low-overhead execution.
- Semgrep introduced overhead on small codebases but scaled more efficiently on large projects thanks to parallelism.
- Execution time was the main driver of energy consumption across tools.
- Both tools showed low variability over 30 runs, confirming reliability of measurements.

5 Limitations and Future Work

- Limited number of tools and open-source projects analyzed.
- Hardware-specific results may not generalize across all setups.
- Future work includes expanding the framework to other programming languages and static analysis tools.

6 Conclusion

This study presents a reproducible framework for evaluating the energy consumption of static analysis tools, emphasizing sustainability as a vital non-functional aspect of secure software development. Through experiments on real-world Python projects, we show that energy usage is shaped not only by the tool itself, but by its architecture, rule complexity, and the nature of the target codebase. Bandit, with its simple, single-threaded design, consistently consumed less total CPU energy on small to mid-sized projects. Its speed and low overhead make it ideal for lightweight, frequent scans. Semgrep, although more power-efficient per second, incurs higher overhead on smaller projects. However, its scalable, parallel architecture and rich ruleset support allow it to perform competitively ,and sometimes more efficiently on large, complex codebases. Our findings suggest that energy-aware tool selection should consider project size and analysis goals. Bandit excels in speed and simplicity, while Semgrep is better suited for deep, policy-driven audits at scale. This insight helps developers align security practices with sustainability, promoting greener software engineering.

References

[1] Mamdouh Alenezi and Mohammad Zarour. 2020. On the relationship between software complexity and security. (2020). https://arxiv.org/abs/2002.07135 arXiv: 2002.07135 [cs.SE].

[2] Brian Chess and Gary McGraw. 2004. Static analysis for security. *IEEE security & privacy*, 2, 6, 76–79.

[3] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. 2019. Greenscaler: training software energy models with automatic test generation. *Empirical Software Engineering*, 24, 1649–1692.

[4] Luis Cruz and Rui Abreu. 2019. On the energy footprint of mobile testing frameworks. *IEEE Transactions on Software Engineering*, 47, 10, 2260–2271.

[5] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2017. Petra: a software-based tool for estimating the energy profile of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 3–6.

[6] [n. d.] Energy-aware software testing. (). https://robertoverdecchia.github.io/papers/ICSE_2025.pdf#:~:text=stated%20that%20%E2%80%9Ctesting%20not%20only,In%20a%20more%20recent.

[7] Abram Hindle. 2015. Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Software Engineering*, 20, 374–409.

[8] [n. d.] Influence of static code analysis on energy consumption of software. (). https://dl.gi.de/server/api/core/bitstreams/0e14dab3-b66b-411e-ad66-05eefae6756d/content.

[9] Sara S Mahmoud and Imtiaz Ahmad. 2013. A green model for sustainable software engineering. *International Journal of Software Engineering and Its Applications*, 7, 4, 55–74.

[10] Gary McGraw. 2004. Software security. *IEEE Security & Privacy*, 2, 2, 80–83.

[11] Martin Otieno, David Odera, and Jairus Ekume Ounza. 2023. Theory and practice in secure software development lifecycle: a comprehensive survey. *World Journal of Advanced Research and Reviews*, 18, 3, 053–078.

[12] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. 2015. What do programmers know about software energy consumption? *IEEE Software*, 33, 3, 83–89.

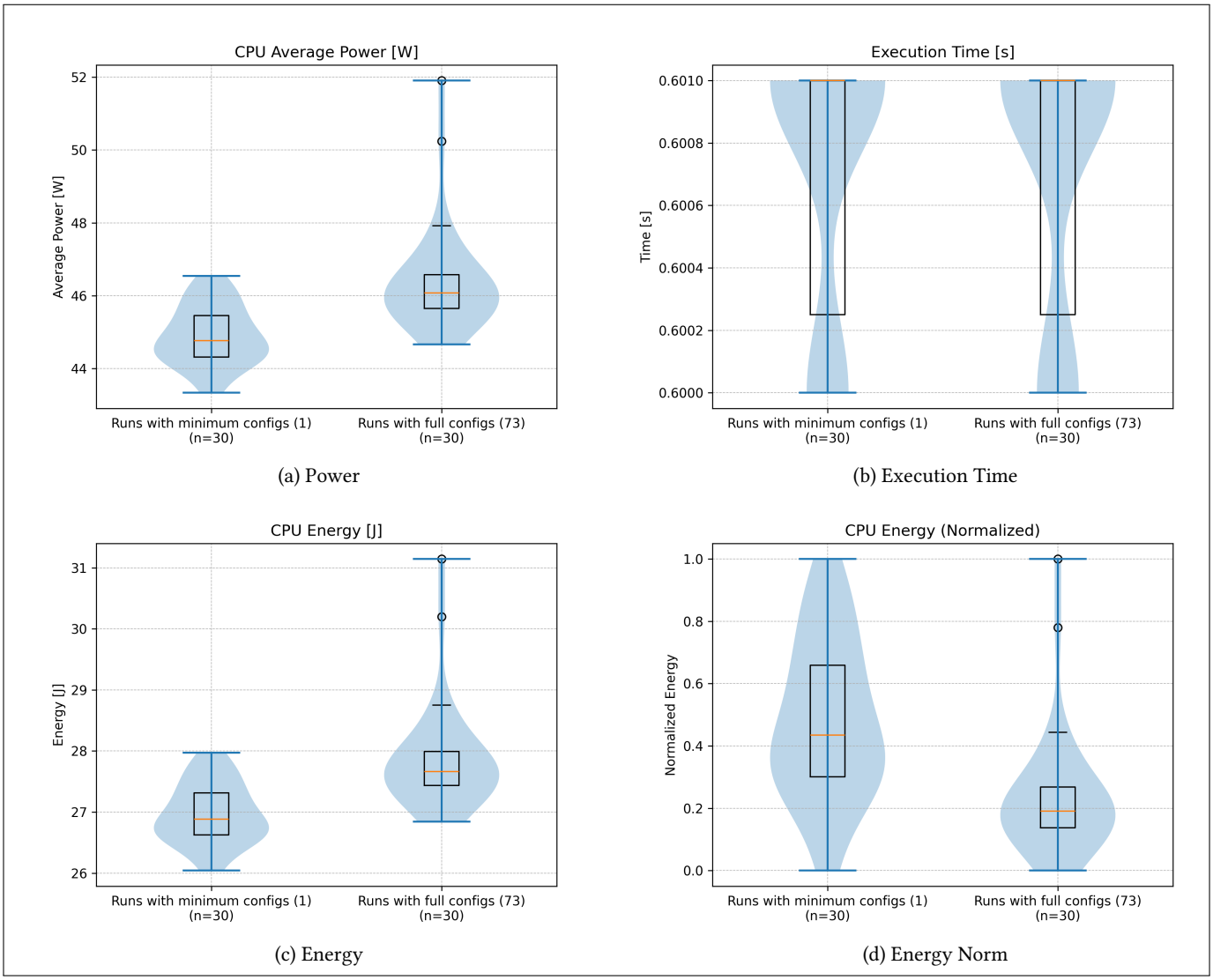
[13] Rami Shehab, Abrar s.alismail, Dr Mohammed Amin Almaiah, Dr Tayseer Alkhdour, Dr Belal Mahmoud AlWadi, and Dr Mahmaod Alrawad. 2024. Assessment of cybersecurity risks and threats on banking and financial services. *J. Internet Serv. Inf. Secur.*, 14, 3, (Aug. 2024), 167–190.

[14] Frederick G Tompkins and Russell S Rice. 1986. Integrating security activities into the software development life cycle and the software quality assurance process. *Computers & Security*, 5, 3, 218–242.

[15] Andy Zaidman. 2024. An inconvenient truth in software engineering? the environmental impact of testing open source java projects. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*, 214–218.

A Appendix

A.1 Performance Plots by Security Tool and Project



(a) Power

(b) Execution Time

(c) Energy

(d) Energy Norm

Figure 4: Performance metrics for `bandit_DeepSeek-V3`.

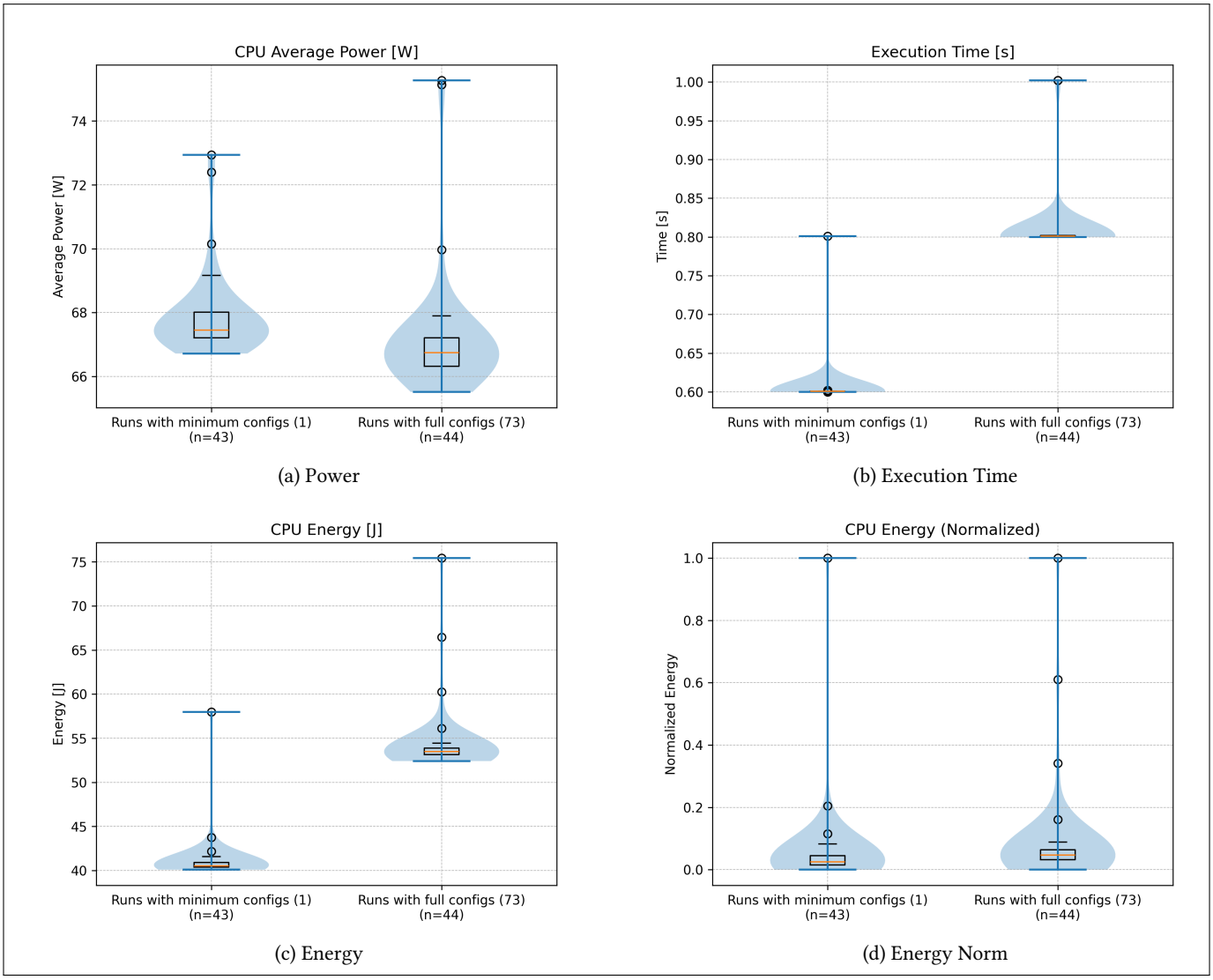


Figure 5: Performance metrics for `bandit_requests`.

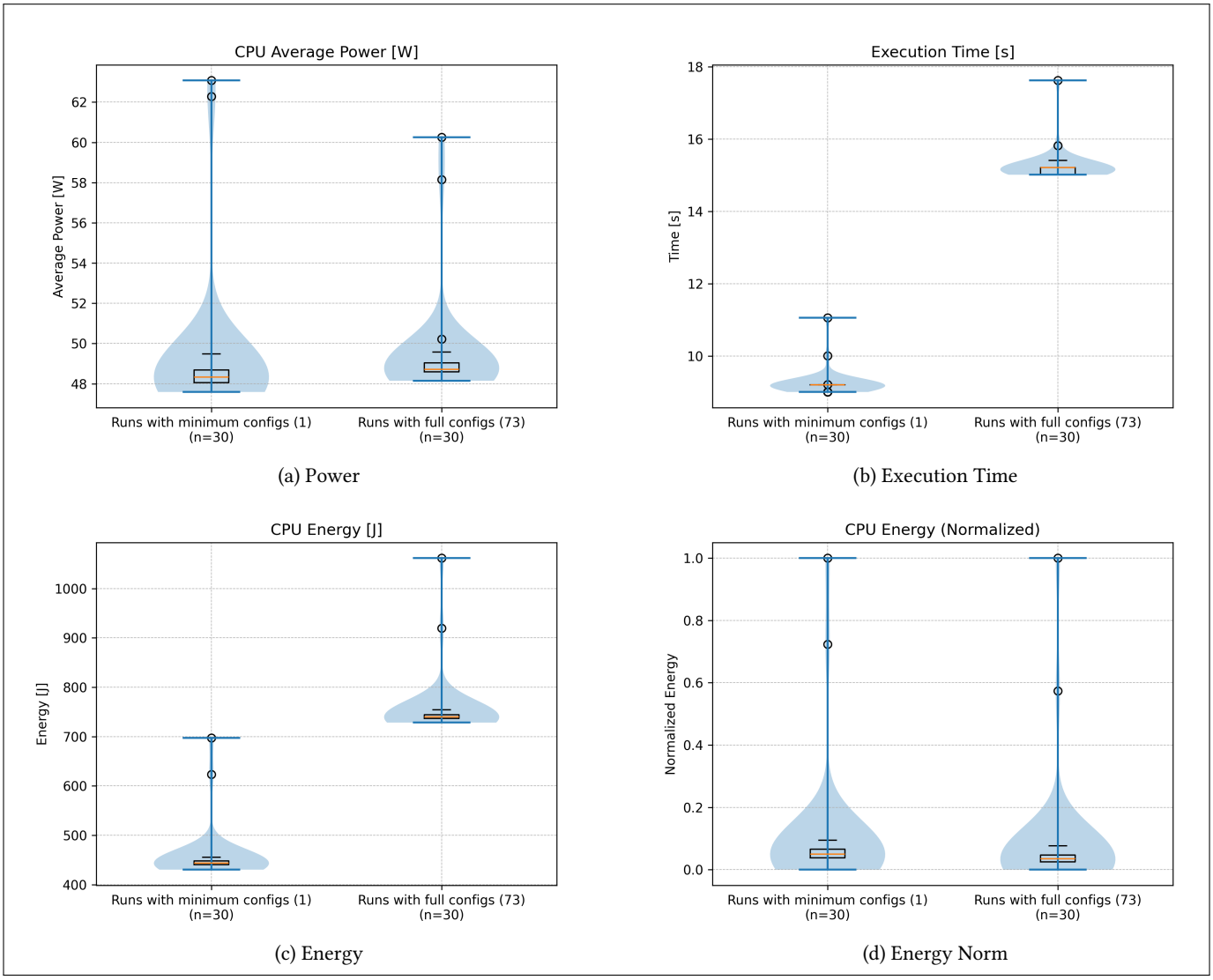


Figure 6: Performance metrics for `bandit_vllm`.

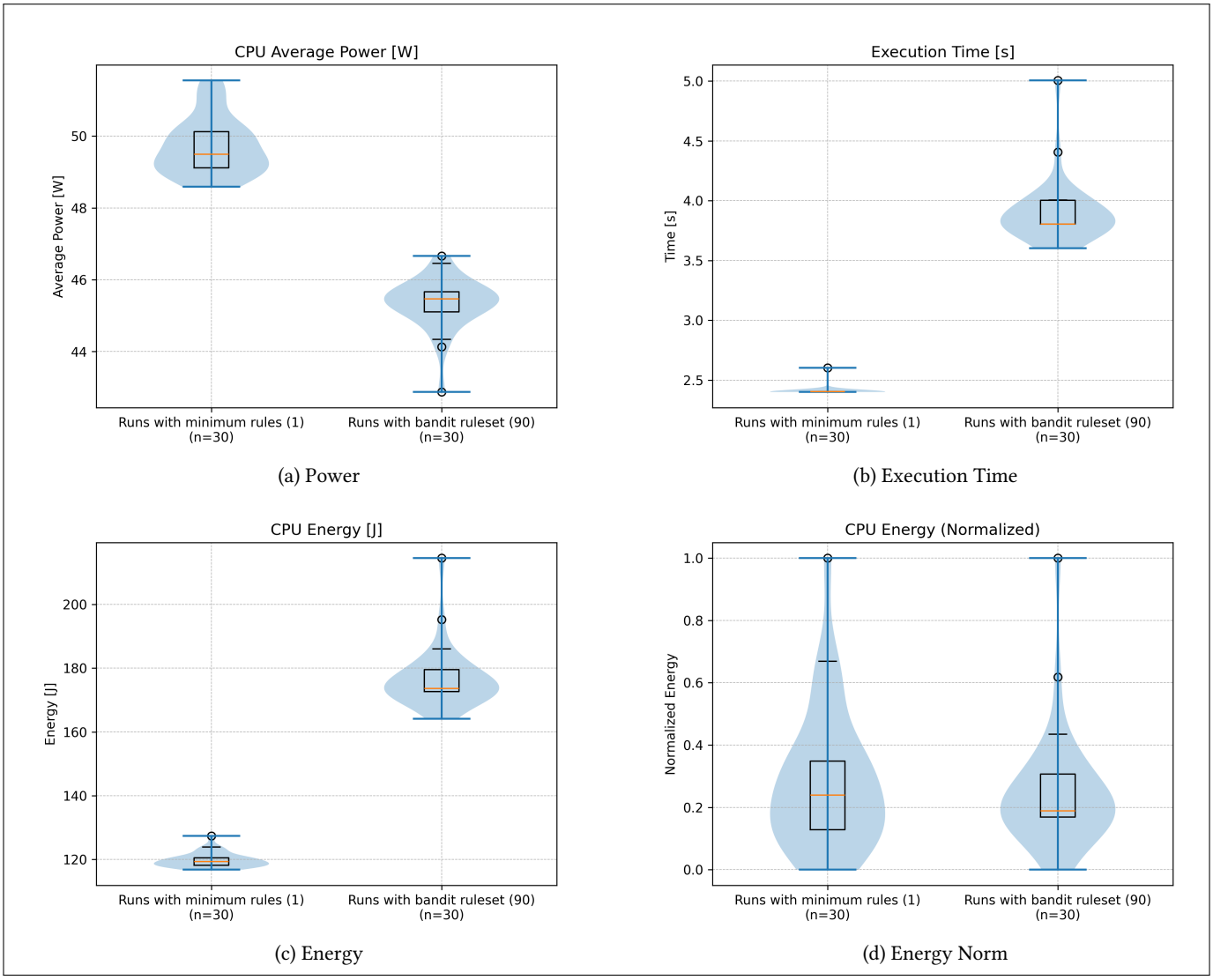


Figure 7: Performance metrics for `semgrep_DeepSeek-V3`.

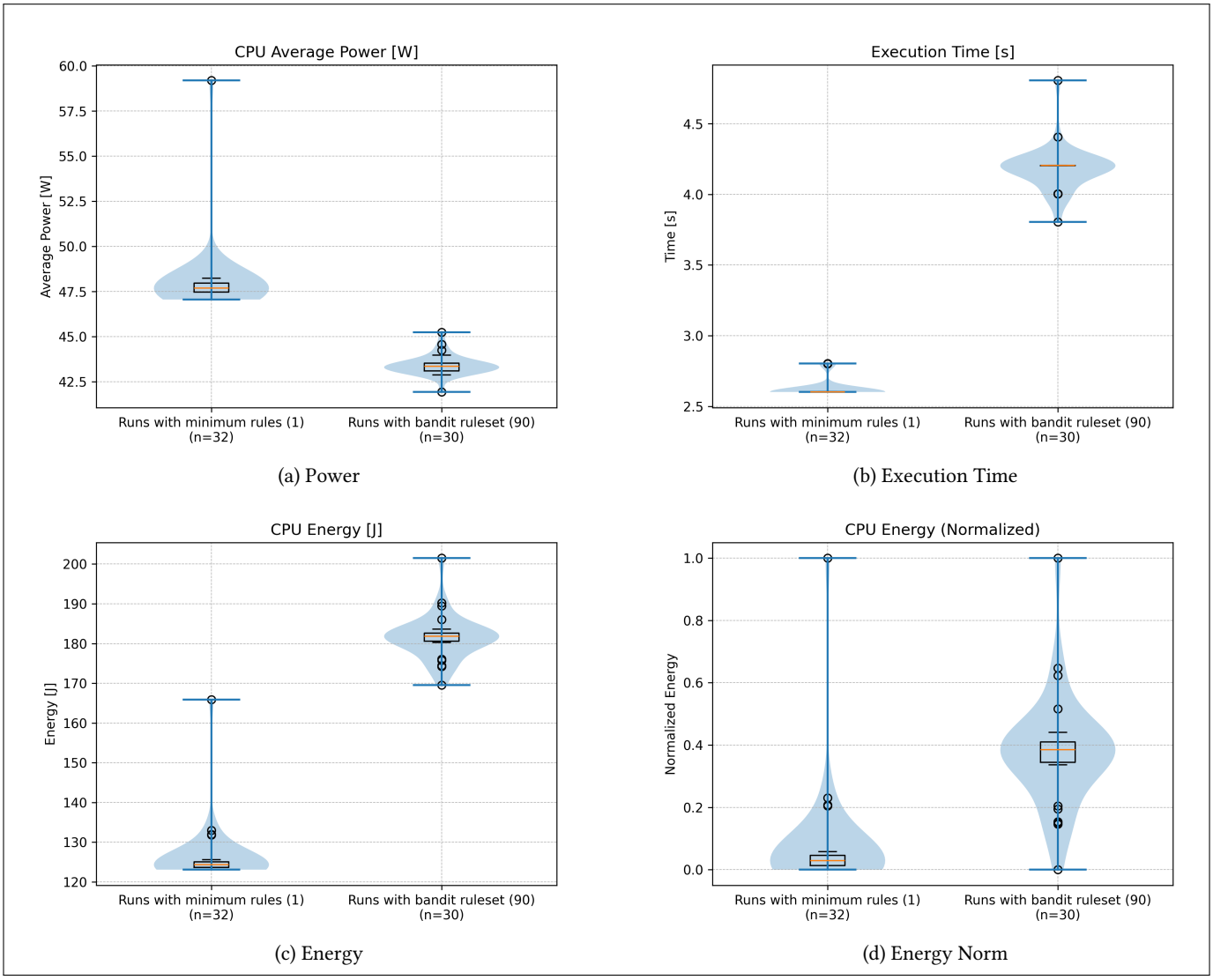


Figure 8: Performance metrics for `semgrep_requests`.

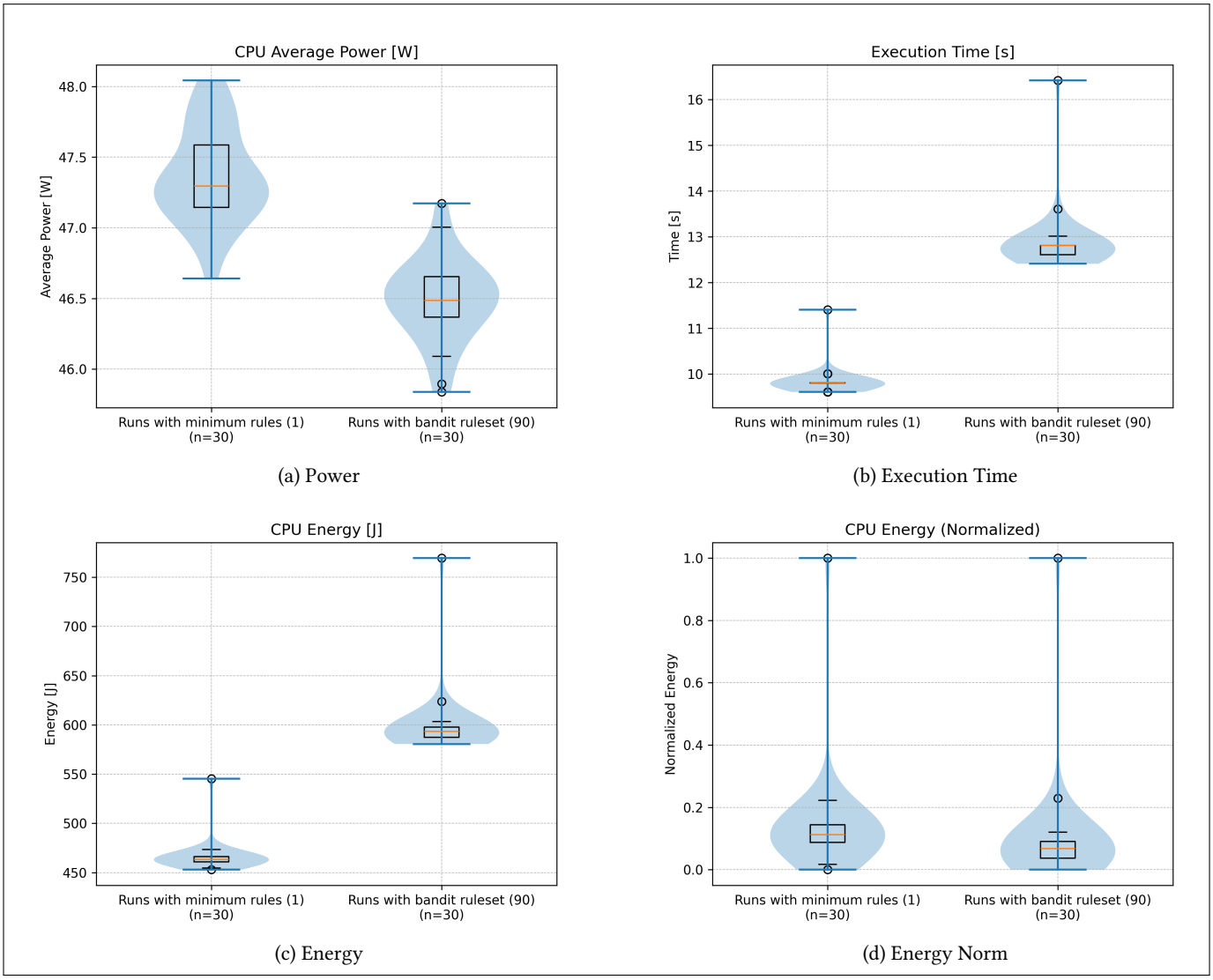


Figure 9: Performance metrics for `semgrep_vllm`.