

Beyond Response Time: Investigating the Power Behavior of Performance Antipatterns

Alessandro Aneggi ✉ 

Free University of Bozen-Bolzano, Bolzano, Italy

Vincenzo Stoico ✉ 

Vrije Universiteit Amsterdam, Amsterdam, The Netherlands

Andrea Janes ✉ 

Free University of Bozen-Bolzano, Bolzano, Italy

Abstract

Background. Performance antipatterns are well known for degrading the responsiveness of microservice-based systems. However, although their impact on latency has been widely studied, their relationship with energy consumption remains largely unexplored.

Aims. This study empirically investigates whether ten widely recognized performance antipatterns, as defined by Smith and Williams, affect power consumption and energy-related behavior in containerized environments.

Method. We implemented ten antipatterns as isolated containerized services and evaluated them under controlled load conditions. For each antipattern, we conducted 30 repeated runs and collected measurements of response time, CPU and DRAM power consumption, and resource utilization.

Results. While all injected antipatterns degraded performance as expected, only a subset showed a statistically significant positive relationship between increased response time and higher power consumption. Some antipatterns exhibited a clear association between performance degradation and increased power demand. In contrast, others reached CPU saturation early, such that power draw remained nearly constant despite rising response times. In these cases, additional slowdowns primarily increased total execution time rather than instantaneous power demand.

Conclusions. Performance antipatterns do not affect power and energy-related behavior uniformly. This study provides a systematic foundation for characterizing how different performance flaws are associated with power and energy-related behavior, offering actionable insights for architects aiming to design more energy-aware software systems.

2012 ACM Subject Classification Software and its engineering → Software design tradeoffs; Software and its engineering → Software design techniques

Keywords and phrases performance antipatterns, microservices, power consumption, energy consumption, energy efficiency

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.30

Category Technical Track Paper

Supplementary Material *Dataset:* <https://doi.org/10.5281/zenodo.21978998> [3]

1 Introduction

Software systems often experience recurring conditions that negatively affect their performance. Recognizing, avoiding, and resolving these issues is vital for maintaining high performance and ensuring software sustainability over time and across contexts. Since their formalization in 2002 by Smith et al. [39], software performance antipatterns have been the subject of extensive study across different domains of software engineering. Current



© Alessandro Aneggi, Vincenzo Stoico, and Andrea Janes;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 30; pp. 30:1–30:20



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

research emphasizes early detection of antipatterns during the software engineering process, especially at the architectural level, to reduce future issues and avoid costly refactoring in production [15, 7, 25, 33, 6, 42].

Despite the recognized importance of antipatterns for performance optimization, *research on their effects on software energy efficiency remains limited*, as also shown by [5]. Architectural decisions, however, can substantially influence energy efficiency [47]. The authors of [47] describe a complex interplay between energy consumption and performance in microservice architectures. Specifically, finer service granularity tends to increase both energy usage and response time compared to coarser designs. Moreover, higher workloads, represented as higher request rates, further amplify both energy consumption and performance impact. The study underscores that the relationship between energy efficiency and performance, shaped by service granularity and system scale, must be carefully evaluated during the design phase.

At the current stage, *it remains unclear whether performance antipatterns are also harmful to software energy efficiency* [26]. Establishing such evidence would be valuable for software architects, enabling them to assess performance-energy tradeoffs during the design phase. Performance indicators may be independent of energy use, with the former being favored in contexts where performance is assessed primarily through output quality or throughput rather than energy efficiency. In this context, *power* represents the instantaneous rate of energy use, whereas total *energy* is obtained by integrating power over the measurement interval. Analysing power separately allows us to determine whether performance degradation is accompanied by an increase in instantaneous demand [4]. However, stable power does not necessarily imply stable energy efficiency, because a slower system may complete less useful work during the same interval.

Our **goal** is to investigate whether and to what extent performance antipatterns are associated with different power-consumption behaviors. We design and conduct a controlled experiment on the ten performance antipatterns described by Connie Smith [40]. These antipatterns are manually implemented as isolated containerized services and profiled under a fixed workload and observation period of 20 minutes. This experimental setup enables us to examine the relationship between performance and energy-related metrics while controlling potential confounding factors, such as thermal variability and background operating system activity. We collect response time, power consumption, and resource utilization, and we use CPU utilization as a control metric. The resource utilization data allow us to verify whether the observed behavior aligns with the antipattern definitions, as the original descriptions provide no implementation guidelines and are intended to remain implementation-agnostic.

Our **results** show that the occurrence of performance antipatterns does not necessarily lead to energy waste. The Ramp, God Class, and Traffic Jam show increases in both response time and power consumption, while also saturating CPU usage. In these scenarios, the prolonged response time can increase the energy required to complete a given amount of useful work, even when instantaneous power remains stable. For the other antipatterns, power consumption is not significantly associated with response time. In such cases, response time remains stable, while power and CPU resources are still unnecessarily drained. The Unnecessary Processing antipattern exemplifies this behavior.

The **main contributions** of this study are as follows: (i) an empirical analysis of the relationship between response time and power consumption in the presence of performance antipatterns; (ii) an in-depth discussion of the findings, emphasizing the trade-offs between energy efficiency and software performance; and (iii) a replication package containing the raw data, analysis scripts, and source code to facilitate full reproducibility of the study (see [3]).

Our findings reveal that performance antipatterns are not inherently detrimental to energy efficiency. Consequently, software architects should carefully assess the trade-offs between performance and energy consumption during design-time. For researchers, the study underscores the importance of treating response time and energy efficiency as independent dimensions of analysis. Finally, our work bridges the two communities, illustrating how insights from software performance evaluation can enrich software architecture design.

2 Related Work

The term *Antipattern* was coined by Andrew Koenig in 1995 to describe solutions to recurring problems that superficially look like good solutions but later reveal disadvantages, making them poor solutions [36]. Smith and Williams [39] proposed a catalog of recurring performance antipatterns, covering processing imbalance, unnecessary work, ramp effects, inefficient database retrieval, excessive parallelism, god classes, excessive allocation, costly lookup chains, serialization bottlenecks, and backlog-related degradation. *Performance antipatterns* have been studied as multi-level software pathologies [38], through design-time models such as Queuing Networks [33, 13], and via runtime multivariate detection for CI/CD pipelines [6].

Software energy consumption can be measured using physical power meters, via on-chip energy counters, or by means of software-only estimators based on analytical or statistical models [23]. The introduction of RAPL in modern CPU architectures has significantly increased the possibility of collecting fine-grained energy data in software and attributing it to specific workloads or software components [19]. Building on this capability, several recent works focus on software, server, and container-level observability of energy and power consumption [32, 1, 16].

The relationship between performance and energy efficiency in server-side systems is non-trivial. From a software engineering perspective, empirical studies show that design decisions impact energy and performance differently: [28] show that design patterns often increase energy consumption despite performance intentions, while refactoring code smells can improve energy efficiency by 49% or increase it by 70% depending on the smell type [43]. [35] demonstrate that best practices making CPU usage more energy-proportional can reduce consumption up to 25%, but parallel execution improves the energy-performance trade-off compared to sequential execution. Across different contexts, researchers find discrepancies between energy and performance metrics. Code smell refactoring shows mixed results [43], while microservice techniques present complex trade-offs: reducing energy by up to 5.6% can increase latency by 25.9% in some cases, while decreasing it by 72.5% in others [45]. Architectural choices also affect these metrics, as demonstrated by microservice granularity studies [47]. These results suggest that performance problems and energy inefficiencies are often correlated but not perfectly aligned, motivating an explicit analysis of software performance antipatterns from the energy-consumption perspective.

Recent research establishes energy as a first-class quality attribute in software architecture, requiring explicit consideration throughout the design process [9]. Foundational work demonstrates that energy-aware architectural abstractions and design principles are essential for system-level optimization. A key observation is that energy-optimal resource allocation configurations often diverge significantly from performance-optimal solutions [46], necessitating dedicated energy-aware resource management strategies. Industrial empirical studies validate the effectiveness of architectural tactics in practice: [17] demonstrate that deployment model selection and granular auto-scaling can achieve energy reductions ranging from 21-50% under typical loads. In microservice contexts, balancing energy efficiency with

failure resilience remains a critical concern [34]. While evidence confirms that energy-aware design is broadly applicable across diverse application domains, adoption barriers persist in industrial practice due to limited measurement transparency and the need for systematic decision-making.

Performance antipattern detection employs multiple approaches across abstraction levels. Model-based frameworks like PADRE [14] identify antipatterns at design time through UML analysis with automated refactoring; runtime statistical methods [6] enable CI/CD integration. *Energy measurement* is based on two main methodologies: monitoring (for CPU, DRAM and GPU) or estimation (which can also include Network and Disk) [31]. Monitoring relies on Intel’s Running Average Power Limit (RAPL) which is a hardware interface built into modern Intel processors that provides fine-grained energy consumption measurements at runtime for CPU, uncore components, and DRAM through model-specific registers [22]. Various tools are available based on RAPL with varying levels of granularity, such as Scaphandre [20], which provides per-process attribution across bare metal and containerized environments; Powerjoular [27], which enables process-level Linux monitoring; PowerLetrics [18], which offers real-time energy footprints; and JoularJX [27], which targets Java-level granularity. However, [26] reveals a critical phenomenon: seven of the studied object-oriented and mobile antipatterns improve energy efficiency of an Android app when removed, while two paradoxically worsen it, indicating that performance optimization and energy efficiency are not perfectly aligned. Despite mature detection tools for performance and a comprehensive measurement infrastructure for energy, no integrated approach combines both perspectives to detect energy-specific antipatterns, particularly in microservice contexts where granularity and resource allocation create novel energy-performance trade-offs.

3 Study Design

This work investigates whether and how much the presence of performance antipatterns impacts energy efficiency beyond degrading system performance. To this end, we analyse how power consumption evolves over time under a controlled workload.

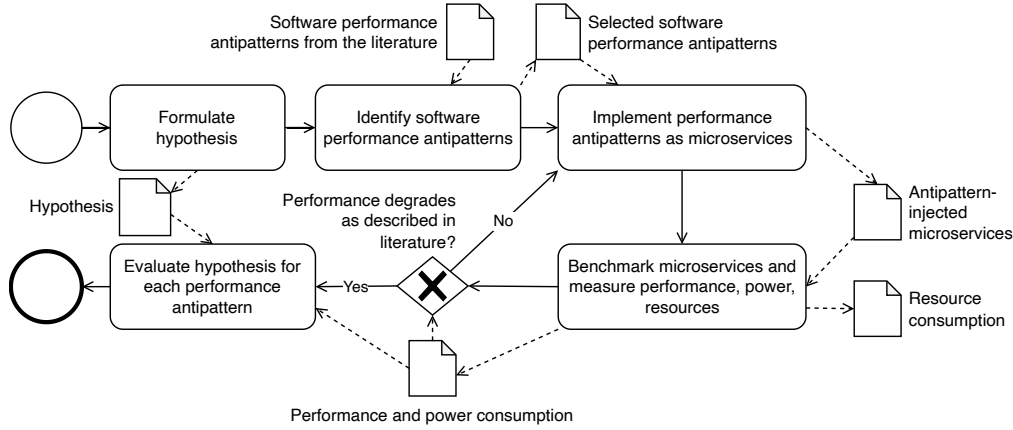
This analysis enables a deeper understanding of power/response time trade-offs by characterizing the degree and conditions of their correlation. When a strong positive relationship is observed, increasing response time is accompanied by increasing power demand. When the relationship is weak or absent, response time cannot be used as a reliable proxy for power behavior. When the correlation is weak or absent, which can occur due to parallelism, performance optimization may be insufficient or even counterproductive for energy efficiency.

Formulated as a GQM goal [10], this study aims to analyze the performance and power-consumption behavior of ten established performance antipatterns for the purpose of characterizing the relationship between performance degradation and instantaneous power demand, with respect to response time, CPU power, DRAM power, throughput, and CPU utilization, from the viewpoint of software architects and software engineering researchers, in the context of isolated containerized applications executed under controlled workloads.

From this reasoning, we derive the following research question: *how does the presence of performance antipatterns affect the power consumption of containerized software?*

Figure 1 depicts the research methodology followed in this study: we begin with hypothesis formulation (see Sect. 3) and the study of the related work (see Sect. 2). As a next step, we identify the software antipatterns we want to evaluate and implement them (see Sect. 4). An empty application was executed under the same conditions as an environmental control to characterize background power consumption and identify temporal effects introduced by

the testbed or measurement setup. For each execution, the initial stable period was used to normalize the temporal evolution of response time and power consumption. We then assess whether the antipattern implementations exhibit the expected performance degradation. Valid implementations are subsequently used to evaluate the hypotheses in Sect. 7.



■ **Figure 1** Research Process (in BPMN [29] notation).

To investigate whether microservices that experience performance degradation also exhibit higher power consumption, we formulate the following hypotheses. The **null hypothesis** (H_0) states that no significant relationship exists between system performance and power usage; in other words, variations in response time do not correspond to meaningful changes in power consumption. The **alternative hypothesis** (H_1) posits a significant relationship, whereby higher response times are associated with increased or decreased power demand.

4 Performance Antipatterns Implementation

This section presents the selected performance antipatterns. We base our implementations on the antipatterns described by Smith and Williams [40], whose catalog is widely recognized in the performance engineering community. To the best of our knowledge, there are no self-contained implementations of these performance antipatterns suitable for controlled experimentation in microservice settings. Existing occurrences are typically embedded in larger systems, introducing confounding factors that hinder isolated analysis.

The implementations follow these principles: execution restricted to one CPU core, implementation as a single endpoint within a Python Flask [30] application, faithful reproduction of the antipattern’s described behavior, and deployment within a single container. A SQLite port of the Northwind database [21] was used only for the Circuitous Treasure Hunt, Excessive Dynamic Allocation, God Class, and Sisyphus Retrieval experiments. In the following, we describe the implementation of each antipattern. Antipattern implementations and diagrams are provided in the replication package.

We implement the ten antipatterns as follows. In **Unbalanced Processing**, redundant transformations and an unbounded in-memory store are placed on one processing path. As a result, concurrent requests are processed sequentially. **Unnecessary Processing** performs CPU- and memory-intensive operations that do not provide a functional benefit. In **The Ramp**, each request searches and sorts a continuously growing collection. Therefore, the processing cost increases with the system state. **Sisyphus DB Retrieval** processes complete database tables before it returns only the requested subset. **More is Less** creates several

CPU-bound threads, for which the coordination and contention overhead is higher than the benefit of parallel execution. **God Class** combines validation, hashing, caching, storage, and database access in one handler. **Excessive Dynamic Allocation** repeatedly creates and discards short-lived containers, strings, and nested data structures. **Circuitous Treasure Hunt** retrieves related data through a sequence of nested database queries. **One-Lane Bridge** holds a global lock during expensive computations. This forces concurrent requests into a waiting queue. Finally, **Traffic Jam** alternates between normal and heavy CPU work and creates a backlog that also affects later requests.

5 Experiment Design

We design the experiment using a *within-subject design* [44], where each subject (in this case, each antipattern implementation) is exposed to different conditions, namely a set of concurrent users in our setting. The number of concurrent users corresponds to our unique **independent variable**, which we set to a single treatment level.

This value is chosen based on CPU usage induced by the number of concurrent users requesting the service. To ensure that the performance antipattern can actually manifest during experimentation, the system must operate under a sufficiently high CPU load. For this reason, we select a number of concurrent users that drives CPU utilization to at least 25% and does not lead to any execution errors. This threshold provides the necessary level of system saturation for revealing behaviors that typically arise under medium or high load, such as slowdowns. For all antipatterns, 50 concurrent users were selected, except Unbalanced Processing (10 users), Unnecessary Processing (30 users), and Traffic Jam (30 users); this choice depends on the available hardware configuration, which is described in Sect. 6.

The considered **dependent variables** are power consumption and performance. Power consumption refers to the power consumed by the container from the CPU and memory, measured in Watts (W). Performance is assessed through response time in milliseconds (ms). CPU usage, memory consumption, disk activity, and network utilization are collected not as performance metrics, but to verify the correctness of the antipattern implementation and to confirm its alignment with the descriptions found in the literature. The tools used in the study, together with their roles and deployment locations, are listed in Tab. 1. The collected metrics are presented in Tab. 2, along with their measurement unit and scope, an indication of whether each metric is directly collected or derived, and whether it is a point or cumulative metric.

Each trial, which includes a specific antipattern and number of concurrent users, is repeated 30 times to ensure measurement consistency and mitigate random variation caused by the execution environment, such as unintentionally active background operating-system processes.

Each experiment consists of a 20-minute test phase. Between experiments, we reset the environment by wiping all Docker containers, restarting the Docker service, waiting for 10 seconds, deploying the solution, and waiting an additional 10 seconds. After each test, we include a 30-second cooldown period. The workload is generated with a spawn rate of 10 users per second. Considering these timings, the total duration of the complete experimental campaign can be expressed as $(20 \text{ min} + \text{setup and cool-down}) \times 10 \text{ antipatterns} \times 30 \text{ repetitions}$, which results in approximately 110 hours (4.58 days) of execution time.

■ **Table 1** Components, Role, and Deployment Location.

Tool	Role	Installation location: <u>Driver</u> , <u>Testbed</u>
PPTAM [8] ^a	Executes test scenarios based on deployment scripts and test configurations.	D
PPTAM Agent ^b	Collects and aggregates data from other tools so that PPTAM can collect it.	T
locust ^c	Generates synthetic API traffic to load the system according to a predefined test script and collects the resulting response times per endpoint.	D
cAdvisor ^d	Provides access to container-level resource usage and performance metrics.	T
Powerjoular [27]	Provides access to per-process power consumption.	T
perf ^e	Provides access to hardware performance counters with configuration for DRAM energy estimation.	T
Docker ^f	Hosts the containerized software under test.	T

^a <https://github.com/pptam/pptam-tool>, ^b <https://github.com/pptam/pptam-agent>

^c <https://locust.io/>, ^d <https://github.com/google/cadvisor>

^e https://web.eece.maine.edu/~vweaver/projects/perf_events/, ^f <https://www.docker.com/>

■ **Table 2** Measurements, units, and their classification.

Tool	Metric	Unit	Host ^a	Container ^a	Endpoint ^a	Direct ^b	Derived ^b	Point ^c	Cumulative ^c
cAdvisor	Memory usage	Byte (b)	×	×	×	×	×	×	×
cAdvisor	R/W Disk	Bytes (b)	×	×	×	×	×	×	×
cAdvisor	R/W Network	Bytes (b)	×	×	×	×	×	×	×
Powerjoular	CPU power consumption	Watt (W)	×	×	×	×	×	×	×
Powerjoular	CPU utilization	Percent (%)	×	×	×	×	×	×	×
perf	DRAM power consumption	Microjoule (μ J)	×	×	×	×	×	×	×
locust	Response time	Millisecond (ms)	×	×	×	×	×	×	×
locust	Failures	Amount	×	×	×	×	×	×	×

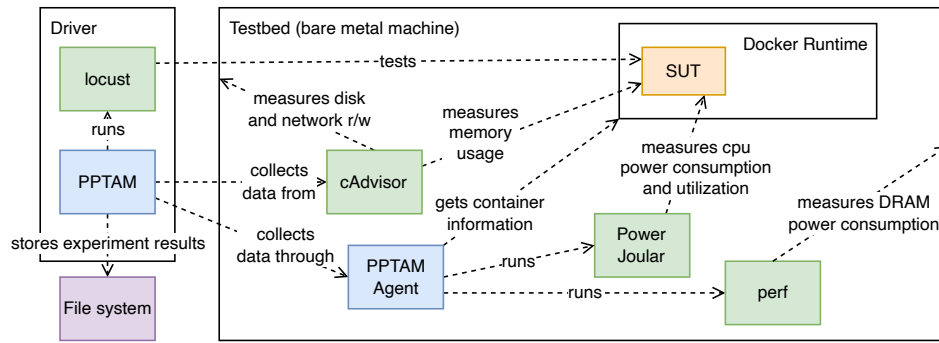
^a Scope dimension, ^b Origin dimension, ^c Temporal dimension

6 Experiment Execution

The experimental infrastructure consists of two separate machines, as illustrated in Figure 2: a *driver* and a *testbed*.

The testbed is a machine equipped solely with the software required for data collection and for executing the software under test. Docker containers are constrained to a single virtual CPU (vCPU) and 1 GB of RAM to ensure consistent resource allocation across all experiments. During each experiment, only the application container is running, and Docker services are restarted between consecutive trials to eliminate residual process states and prevent measurement contamination. The number of concurrent users is ramped up to the target count in under two minutes, establishing a steady load. Following standard performance-measurement practice, data collected during the first two minutes (the warm-up period) are excluded from the analysis to allow the system to reach a stable state.

For the baseline, we deploy the minimal endpoint in a container and drive it with 50 concurrent users, using the same spawn rate and duration as in the other experiments. We record power traces for the same duration as a standard trial.



■ **Figure 2** Container diagram of the experimental infrastructure (C4 notation [12]). Blue containers indicate data collection and experiment coordination containers; green containers indicate data collection tools; orange containers indicate the software under test (SUT); and violet containers indicate external systems.

We operationalize the dependent variables from Sect. 5 by collecting the metrics listed in Tab. 2. Response time is measured in milliseconds (ms), CPU usage in percent, memory, disk and network usage in bytes, later converted to mebibytes (MiB) or gibibytes (GiB).

The **testbed** is a Lenovo X1 Yoga 3rd Gen with an Intel Core i5-8250U CPU and 8 GB of RAM. The software stack consists of Ubuntu 24.04 LTS, Powerjoular 0.4, cAdvisor 0.53, perf 6.8.12, Docker 29.0.2, and the PPTAM *agent* used to collect measurements. The ambient room temperature during the experiments was approximately 25°C, with a variability of $\pm 2^\circ\text{C}$.

We measure power consumption using two complementary tools: Powerjoular for process-level CPU power consumption and perf for DRAM energy consumption. Both tools derive their measurements from Intel RAPL counters exposed through the Linux powercap interface [41]. Power readings are collected at 1-second intervals throughout each trial, with CPU power and DRAM power expressed in Watts (W).

At runtime, we convert consecutive energy readings from perf into average power by computing the change in energy over the corresponding time interval. This yields the average power between the two readings, expressed in Watts.

CPU utilization is reported as a fraction of the total system CPU capacity. Containers are limited to a single vCPU in Docker. Since the machine has four CPU cores, a value of 0.25 corresponds to 100% utilization of a single vCPU. The CPU frequency governor is set to *powersave* to reduce variability caused by frequency scaling.

The **driver** machine is a virtual machine running on Proxmox VE 8.4.14 with 8 vCPUs and 20 GB of RAM. The software installed on this VM includes Ubuntu 24.04 LTS, PPTAM to orchestrate tests and collect data, and Docker 29.0.2. PPTAM internally uses locust.io 2.42.2. During the experiments, the VM has exclusive access to its allocated resources.

On both machines, unnecessary services and background processes are identified and disabled, and network access is limited to minimize external interference and measurement noise.

7 Data Analysis

Our analysis proceeds in five steps. First, we conduct a sanity check through trace visualization by plotting the three measurement groups (performance, energy consumption, and resource utilization) for each experiment. This enables us to visually inspect the data, detect

measurement anomalies, and verify that each antipattern exhibits the characteristic behavior reported in prior work. Second, we perform a descriptive analysis by computing mean, minimum, and maximum values of power consumption and response time for each antipattern and generating box plots to obtain an initial view of their distributions.

Third, we compare each antipattern against the baseline using run-level distributions of power consumption. Since Shapiro-Wilk tests indicate non-normality for several condition-metric pairs, we use Wilcoxon rank-sum tests with Holm correction. The complete baseline-comparison results, including percentage changes with respect to the baseline, are reported in the replication package.

We then assess the association between response time and power consumption through Pearson and Spearman correlation coefficients, capturing both linear and monotonic relationships. Finally, we fit multiple linear regression models for each antipattern experiment, with CPU or DRAM power as the dependent variable. The CPU model includes response time, request rate, CPU utilization, and the interaction between response time and CPU utilization. The DRAM model additionally controls for memory usage. Response time and CPU utilization are mean-centered before computing the interaction term. Using heteroskedasticity-robust (HC3) standard errors, we test whether the response-time and interaction coefficients differ significantly from zero at $\alpha = 0.05$.

8 Results

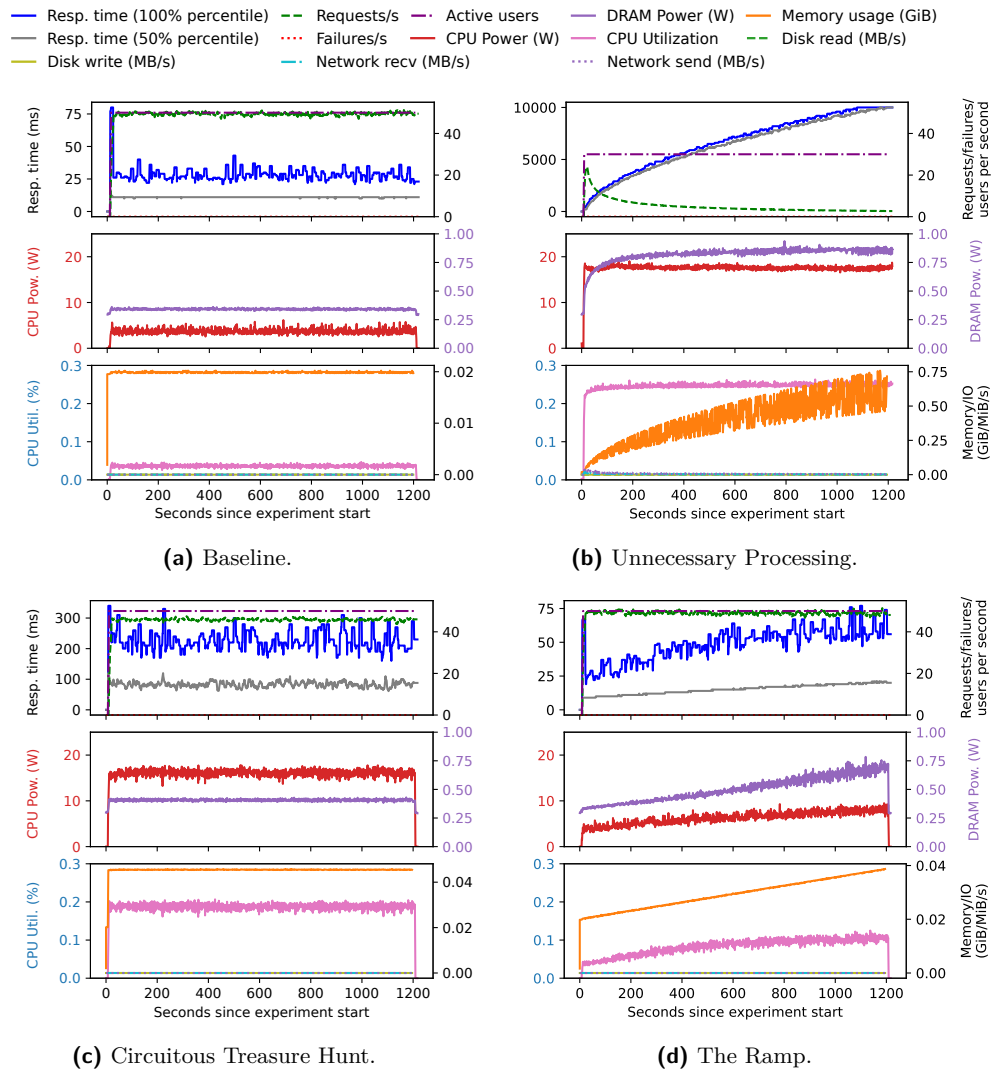
Beginning with *Sanity Check via Trace Visualization*, in Fig. 3 we illustrate examples of collected metrics during the respective experiments. The complete set of plots for every antipattern is available in the replication package [3].

The baseline serves two purposes. First, it supports measurement stability assessment, as CPU and DRAM power remain approximately constant over time with no systematic trend across the 30 runs, as shown in Figure 3a. Second, it provides a minimal reference condition for comparing the power consumption observed under each antipattern.

According to our experimental design, the failure rate remains at zero over the entire execution time, and the CPU utilization reaches at least 0.075, which in our testbed corresponds to approximately 25% of a single vCPU. In Figure 3b, we observe that even with a significant drop in requests per second, no corresponding increase is observed in the number of errors. This substantial reduction in throughput therefore has no impact on the error rate, and for this reason the results of this experiment are considered valid. During the Circuitous Treasure Hunt antipattern (Figure 3c), the response time fluctuates, while the error rate remains zero. We also observe a small fluctuation in the number of requests per second, which is consistent with the expected behavior of this antipattern.

The Ramp antipattern in Figure 3d exhibits the characteristic pattern of steadily increasing metric values, which is precisely how this antipattern is expected to manifest. Nevertheless, in this case as well, both the error rate and CPU utilization satisfy our validity conditions. These observations collectively support the faithfulness of our Python implementation of the considered antipatterns.

Proceeding to *Descriptive analysis*, Table 3 summarizes the average, minimum, and maximum values of response time, CPU power, and DRAM power consumption across all antipattern experiments. The results reveal substantial variability in response time characteristics. Several antipatterns, such as Unbalanced Processing, God Class, and Excessive Dynamic Allocation, exhibit high average response times (7–14 s), while The Ramp and



■ **Figure 3** Trace Visualization of Unnecessary Processing, Circuitous Treasure Hunt, and The Ramp.

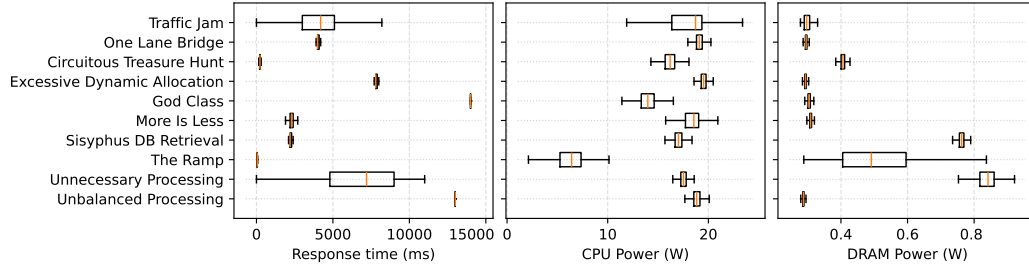
Circuitous Treasure Hunt show response times that remain well below 300 ms. Maximum response time values follow the same pattern, with the most severe antipatterns reaching up to 17 s.

CPU power shows a narrower range across experiments. Mean CPU load typically falls between 14–19 W, with peak values consistently reaching 22–25 W, indicating that most antipatterns drive the system close to full CPU usage during the workload’s high-load phases.

DRAM power consumption exhibits moderate variability. Mean DRAM power remains low for most experiments (approximately 0.29–0.40 W), but some antipatterns such as Unnecessary Processing and Sisyphus DB Retrieval show markedly higher average values (above 0.70 W). Maximum DRAM power ranges from 0.43 W (More Is Less) up to 1.65 W (Unnecessary Processing), suggesting that certain antipatterns cause more pronounced memory-side energy demands.

■ **Table 3** Average, minimum, and maximum values of power consumption and response time for each antipattern. Minimum response time and minimum CPU omitted since they are always 0.

Experiment	Response time (ms)		Power (W)					
	$\bar{x}$	$x_{\max}$	CPU		DRAM			
			$\bar{x}$	$x_{\max}$	$\bar{x}$	$x_{\min}$	$x_{\max}$	
Unbalanced Processing	12833	14000	18.74	23.98	0.29	0.28	0.91	
Unnecessary Processing	6728	11000	17.39	22.51	0.83	0.29	1.65	
The Ramp	49	94	6.21	12.18	0.50	0.29	1.32	
Sisyphus DB Retrieval	2185	2500	16.81	22.40	0.76	0.29	1.07	
More Is Less	2289	2800	18.08	23.26	0.31	0.29	0.43	
God Class	13693	17000	13.93	22.41	0.30	0.29	0.62	
Excessive Dyn. Allocation	7747	8200	19.36	24.74	0.29	0.28	0.69	
Circuitous Treasure Hunt	229	390	15.92	22.10	0.40	0.29	1.34	
One Lane Bridge	3985	4600	18.88	23.86	0.30	0.28	1.20	
Traffic Jam	4139	10000	17.48	23.40	0.30	0.28	1.05	



■ **Figure 4** Response time, CPU and DRAM power box plots across all 30 runs for every studied antipattern; boxes show median/IQR, whiskers show central spread, mean marked (outliers hidden).

Overall, the results indicate that while CPU utilization is relatively similar across antipatterns, response time and DRAM power consumption vary considerably, suggesting that different performance antipatterns impose distinct types and intensities of resource stress on the system. Fig. 4 illustrates the box plots across all experiments for response time, CPU power, and DRAM power.

The results of the *Correlation analysis* are reported in Tab. 4. Pearson’s correlation measures the linear association between two variables, indicating how strongly they vary in a proportional, straight-line manner. Spearman’s rank correlation captures monotonic relationships, detects consistent increasing or decreasing trends even when they are nonlinear, and is less sensitive to outliers. Using both metrics provides complementary insight: agreement between them suggests a stable relationship, whereas divergence indicates potential non-linearity or the influence of extreme values. We indicate agreement using underlined values.

Across the experiments, the correlations between response time and power consumption vary substantially, reflecting the heterogeneous behavior of different antipatterns. Several cases (including The Ramp, Unbalanced Processing, and Excessive Dynamic Allocation) show positive CPU–response time correlations, consistent with increased computational effort accompanying higher response times. In contrast, DRAM correlations are more mixed: some antipatterns (e.g., Unnecessary Processing) exhibit strong positive associations, while others show weak or even negative relationships, indicating that memory activity is not uniformly coupled with response time across patterns. Agreement between Pearson and

■ **Table 4** Pearson and Spearman correlations between CPU and DRAM power and response time for each experiment (30-run aggregates); Underlined values indicate that both correlations share the same sign.

Experiment	CPU		DRAM	
	Pearson r	Spearman ρ	Pearson r	Spearman ρ
Unbalanced Processing	0.797	0.093	-0.113	-0.072
Unnecessary Processing	0.204	-0.101	0.352	0.786
The Ramp	0.664	0.691	0.721	0.748
Sisyphus DB Retrieval	0.613	-0.264	0.641	-0.264
More Is Less	0.526	-0.094	0.112	-0.018
God Class	0.587	0.045	0.068	0.043
Excessive Dyn. All.	0.743	0.062	-0.108	-0.028
Circuitous Treasure Hunt	0.347	0.013	0.261	0.018
One Lane Bridge	0.648	0.029	-0.049	-0.040
Traffic Jam	0.296	0.226	-0.096	-0.158

Spearman coefficients (highlighted by underlined values) suggests that, for those experiments, the relationship is both monotonic and directionally stable, whereas divergences point to non-linear or irregular interactions between performance degradation and power usage.

At this point we proceed to *Regression modeling and hypothesis testing*. We estimate the relationship between response time and power consumption using two ordinary least squares (OLS) regression models for each antipattern experiment, aggregating data across 30 repeated runs. OLS quantifies how a dependent variable changes as a linear function of a set of predictors, such that each regression coefficient represents the average marginal effect of its predictor while holding the remaining variables constant. Both models include an intercept and are fitted only on steady-state samples, excluding a 120-second warm-up interval and ignoring negative power readings. This filtering ensures that transient initialization effects and instrumentation artifacts do not bias the estimated relationship between response time and power consumption, allowing the models to capture behavior that is representative of the system’s stable operating mode.

To account for the possibility that the association between response time and power depends on processor load, we include a response-time $\times$ CPU-utilization interaction term. Response time and CPU utilization are mean-centered within each experiment before constructing the interaction. Consequently, the response-time main effect represents its association with power at the experiment’s mean CPU utilization, while the interaction coefficient indicates how this association changes as CPU utilization increases.

Because system-level performance measurements are typically noisy and exhibit load-dependent variability (see e.g., [37]), the assumption of homoskedastic residuals (i.e., constant variance in the regression errors) is unlikely to hold. In line with this expectation, the Anderson-Darling test [2] rejected normality for all experiments; the Breusch-Pagan test [11] rejected the null hypothesis of homoskedasticity for every antipattern in at least one of the two regression models (CPU or DRAM power). Specifically, all antipatterns except *Unnecessary Processing* showed heteroskedasticity in the CPU-power model, while for DRAM power the null was rejected for all but *More Is Less*, *God Class*, *Excessive Dynamic Allocation*, and *One Lane Bridge*. Thus, each antipattern showed evidence of heteroskedasticity in one or the other analysis.

As a consequence, we employ HC3 heteroskedasticity-consistent (robust) standard errors [24]. HC3 adjusts the estimated uncertainty of regression coefficients in the presence of outliers or non-uniform noise, yielding more reliable inference than classical OLS standard errors when homoskedasticity does not hold. Using HC3 enables valid hypothesis testing even when measurement variability increases over time.

For the CPU power model, we regress CPU power on mean-centered response time, request rate, mean-centered CPU utilization, and their response-time $\times$ CPU-utilization interaction. These predictors are chosen to isolate the contribution of response time to CPU energy usage while accounting for factors that directly influence CPU power. Request rate captures the volume of work offered to the system, and CPU utilization reflects the degree of processor saturation; both variables typically rise and fall with workload intensity and may correlate with response time. By including them, the model controls for load-induced variation in CPU power, ensuring that the estimated response time coefficient represents the effect of performance degradation rather than changes in system demand or CPU occupancy. The goal is thus to obtain an estimate of whether higher response time is associated with increased CPU power consumption under otherwise comparable conditions.

For the DRAM power model, we use the same predictors and interaction term and additionally control for memory usage. The same workload controls (request rate and CPU utilization) are included to account for load-induced changes in memory traffic, while memory usage is added because it directly determines the volume of active DRAM state and is a primary driver of DRAM power draw. Including these predictors addresses potential confounding: DRAM activity may rise due to increased memory pressure or higher throughput, independently of response time. By selecting variables with clear mechanistic relevance to DRAM behavior, the model remains parsimonious while enabling a more accurate assessment of whether increased response time, after controlling for workload and memory demand, is associated with higher DRAM power consumption.

For each experiment (with sample sizes $N > 32,000$ in all cases), we report the response-time coefficient and the response-time $\times$ CPU-utilization interaction coefficient with their HC3 p -values. Because the predictors are mean-centered, the response-time coefficient describes its association with power at average CPU utilization. A significant interaction indicates that this association changes with CPU utilization.

The goal of this analysis is to assess whether increased response time corresponds to higher instantaneous power and whether this relationship depends on CPU utilization. A significant positive interaction indicates that the response-time–power relationship becomes more positive as CPU utilization increases; a significant negative interaction indicates that it becomes less positive or more negative. The detailed results are shown in Tab. 5.

The relationship between response time and power depends on CPU utilization for only a subset of antipatterns. For CPU power, this relationship is significant for Unnecessary Processing, The Ramp, God Class, and Traffic Jam. For DRAM power, it is significant for The Ramp, God Class, Circuitous Treasure Hunt, and Traffic Jam. For the remaining cases, we find no statistically significant evidence that CPU utilization modifies the response-time–power relationship.

These results provide a more nuanced interpretation of the relationship between performance and power. For Unnecessary Processing and The Ramp, the positive interaction indicates that the response-time–power association becomes more positive as CPU utilization increases. For God Class and Traffic Jam, the negative interaction indicates that this association weakens as CPU utilization rises, which is consistent with power approaching a saturation region while response time can continue to increase. Circuitous Treasure Hunt

■ **Table 5** Regression results for CPU and DRAM power. For each power metric, the decision tests whether response time is associated with power at mean CPU utilization. The interaction term tests whether this association changes with CPU utilization.

Experiment	Metric	β_{lat}	CI low	CI high	p_{lat}	$\beta_{lat \times CPU}$	$p_{lat \times CPU}$	Decision
Unbalanced Processing	CPU	-0.000082	-0.000147	-0.000018	0.012364	0.000538	0.956072	Reject ↓
	DRAM	0.000000	-0.000000	0.000001	0.799539	0.000005	0.945942	Keep
Unnecessary Processing	CPU	-0.000006	-0.000013	0.000000	0.068710	0.000725	0.008508	Keep
	DRAM	-0.000004	-0.000006	-0.000002	0.000397	-0.000019	0.759420	Reject ↓
The Ramp	CPU	0.019124	0.018130	0.020118	0.000000	0.425367	0.000000	Reject ↑
	DRAM	-0.000245	-0.000303	-0.000187	0.000000	0.073638	0.000000	Reject ↓
Sisyphus DB Retrieval	CPU	-0.000113	-0.000195	-0.000031	0.007143	0.047599	0.066651	Reject ↓
	DRAM	-0.000005	-0.000007	-0.000002	0.000154	-0.000293	0.276883	Reject ↓
More Is Less	CPU	-0.001141	-0.001293	-0.000989	0.000000	0.003680	0.956921	Reject ↓
	DRAM	-0.000001	-0.000002	-0.000001	0.000007	-0.000009	0.201629	Reject ↓
God Class	CPU	0.000189	0.000153	0.000226	0.000000	-0.008496	0.020297	Reject ↑
	DRAM	-0.000000	-0.000000	0.000000	0.134119	-0.000029	0.003702	Keep
Excessive Dyn. All.	CPU	0.000017	-0.000026	0.000059	0.436476	0.018740	0.456387	Keep
	DRAM	-0.000000	-0.000001	0.000001	0.973993	-0.000036	0.215688	Keep
Circuitous Treasure Hunt	CPU	-0.000566	-0.000771	-0.000360	0.000000	-0.059339	0.098669	Reject ↓
	DRAM	0.000005	-0.000001	0.000012	0.118278	0.000779	0.033509	Keep
One Lane Bridge	CPU	-0.000046	-0.000119	0.000026	0.209550	-0.005516	0.815069	Keep
	DRAM	0.000000	-0.000001	0.000002	0.463445	-0.000009	0.447557	Keep
Traffic Jam	CPU	0.000031	0.000025	0.000037	0.000000	-0.002154	0.000000	Reject ↑
	DRAM	-0.000000	-0.000000	0.000000	0.800827	-0.000004	0.023013	Keep

exhibits this conditional relationship only for DRAM power, suggesting that its memory-side behavior depends on processor load even though its CPU-power interaction is not significant. Thus, response time does not have a constant association with power across all utilization levels, reinforcing our overall conclusion that the coupling between response time and power is antipattern-dependent.

9 Discussion

Our results show that antipatterns affect power consumption non-uniformly. Increases in power are mainly driven by workload intensity rather than response time alone. We observe that power consumption growth is primarily influenced by workload intensity rather than by the accumulated execution time alone. For instance, the Circuitous Treasure Hunt antipattern, illustrated in Figure 3c, quickly increases CPU utilization to around 80%, capping power draw at an upper bound, even as response time holds steady in a range below 300 ms.

This case shows that increasing *response time is not accompanied by increasing instantaneous power*. However, the resulting energy efficiency also depends on the amount of useful work completed during the measurement interval.

Consistent with this profile, the regression analysis in Table 5 provides no significant evidence that CPU utilization changes the relationship between response time and CPU power.

The Circuitous Treasure Hunt scenario therefore illustrates that response time and instantaneous power do not necessarily change together.

Despite a CPU usage spike of around 80 %, this antipattern is not primarily CPU-intensive. The observed response time is therefore likely influenced by its implementation. Specifically, our implementation performs multiple sequential database calls, which primarily stress disk and memory resources rather than the CPU. As a result, the way data is retrieved and joined across different tables has a major impact on response time. Optimizing query batching or introducing asynchronous requests could further reduce unnecessary CPU activity while maintaining the intended functionality and response time characteristics of the scenario.

This behavior differs from that of Unnecessary Processing, shown in Figure 3b. Like Circuitous Treasure Hunt, it operates close to CPU saturation and exhibits a relatively stable overall power profile. However, the significant positive interaction indicates that the response-time–CPU–power association becomes more positive as CPU utilization increases. However, response time increases steadily throughout the observation period. The system therefore exhibits stable power after reaching CPU saturation, while response time continues to increase. Unlike Circuitous Treasure Hunt, this antipattern performs CPU-intensive operations, repeatedly carrying out redundant hash computations and input validations. It also continuously allocates and reorders data in memory, resulting in a sequence of superfluous operations whose footprint gradually expands over time. This ongoing growth in unnecessary processing causes increasing response time, resembling a mild instance of The Ramp antipattern. This observation suggests that, depending on the implementation, *multiple antipatterns may manifest simultaneously*.

Our implementation of The Ramp also provides additional insights for practitioners. Figure 3d shows a typical Ramp situation: both CPU and DRAM power consumption increase steadily over time, and this trend is mirrored by the growth in response time. By definition, The Ramp antipattern is characterized by a workload that becomes progressively heavier during the observation window. In our case, as more data are accumulated in memory, each new request must process a larger in-memory dataset. In the concrete implementation, this behaviour is caused by continuously storing additional data structures and then performing search and ordering operations on them. As the in-memory collections grow, these operations become increasingly expensive, both in terms of computation and memory access. Consequently, the application performs more work per request, leading to longer response times and a gradual increase in CPU and DRAM power consumption, consistent with the expected profile of The Ramp antipattern. The significant positive interactions for both CPU and DRAM power further indicate that this coupling becomes stronger as CPU utilization increases. The pattern implies long-term scalability and capacity issues: since each request becomes more expensive as the system accumulates data, the system will not only slow down but also show decreasing throughput over time, which is a defining characteristic of The Ramp antipattern. This means that adding hardware itself will not solve the problem sustainably, because the fundamental per-request work keeps increasing with dataset size. *This indicates that refactoring data structures or choosing algorithms whose computational complexity grows more slowly with dataset size, such as indexed access, partitioning, or batching, is at least as important as restructuring the workload over time.*

We suggest that practitioners prioritize those antipatterns that are associated not only with degraded system performance, but also with increased power consumption. In particular, *The Ramp* deserves attention because it increases energy usage over time as the system state grows. Although this behaviour may not be entirely removed in every case, optimizing the corresponding code path remains worthwhile to improve the sustainability of the software. *Excessive Dynamic Allocation* also represents an important target, as the repeated creation and disposal of short-lived objects increases memory activity and contributes to additional

CPU and energy demand over time. A similar concern emerges for *Traffic Jam*, where energy consumption increases during phases of backlog accumulation and recovery, reflecting the extra overhead imposed by congestion and repeated scheduling activity rather than productive work. Finally, *God Class* is particularly relevant from a design perspective, since the concentration of multiple responsibilities may amplify inefficiencies similar to those observed in other antipatterns, thereby contributing to both poor performance and increased energy consumption. More generally, our results suggest that optimization strategies should not focus on response time alone. Some antipatterns exhibit relatively stable power profiles despite substantial performance degradation, whereas others show a clearer coupling between latency and power demand. For this reason, practitioners should evaluate the trade-offs among response time, CPU usage, and power consumption in order to identify hidden operational computational complexity more effectively.

We observe that the fixed-duration executions show different total energy measurements across antipattern implementations. Although total energy consumption is not the primary focus of this study, it depends on contextual factors such as the number of users, the hardware platform, and the specific implementation of each antipattern. This work seeks to examine the relationship between power consumption, response time, and resource utilization. Notably, our findings indicate that antipatterns executed under similar conditions exhibit distinct energy loads, *offering a promising foundation for further investigation into the energy implications of performance antipatterns*. Table 6 presents the average energy consumption associated with each antipattern, computed by applying the trapezoidal rule to the power measurements. We notice that antipatterns executed with the same number of concurrent users (50), that is, all except Unbalanced Processing (10), Unnecessary Processing (30), and Traffic Jam (30), demonstrate different CPU and DRAM consumption. Excessive Dynamic Allocation results in the highest CPU energy usage, while Unnecessary Processing yields the greatest DRAM energy load. These findings are valuable for practitioners evaluating whether mitigating a given antipattern can help achieve a more balanced trade-off between energy efficiency and performance. For instance, God Class and The Ramp introduce significantly lower energy overhead compared to the other antipatterns.

■ **Table 6** Average CPU and DRAM energy consumption.

Experiment	Energy (kJ)		Experiment	Energy (kJ)	
	CPU	DRAM		CPU	DRAM
Unbalanced Processing	21.93	0.31	God Class	17.95	0.33
Unnecessary Processing	20.52	0.90	Excessive Dynamic Allocation	22.72	0.32
The Ramp	10.62	0.56	Circuitous Treasure Hunt	19.25	0.44
Sisyphus DB Retrieval	19.95	0.83	One Lane Bridge	22.22	0.32
More Is Less	20.98	0.33	Traffic Jam	20.83	0.32

10 Threats To Validity

Internal Validity. Each performance antipattern is manually implemented and may introduce unintended effects. We mitigate this threat by visually validating the behavior of every antipattern against its description in [40] using line plots of response time, CPU utilization, and power consumption (see Fig. 3d). Between each experiment, applications containing the antipattern are uninstalled, the entire Docker environment is restarted, and the applications are reinstalled.

External Validity. Our results are obtained from Python-based containers running on a single hardware configuration equipped with an Intel Core i5-8250U CPU. This controlled setup improves reproducibility but limits generalizability to other platforms, languages, and deployment environments. Real-world systems often exhibit more heterogeneous workloads and resource variability, which may influence the observed effects. We mitigate this risk by providing the implemented antipatterns together with all collected data in the form of a replication package [3].

Construct Validity. Because [40] describes performance antipatterns only conceptually, their exact implementations are not specified. We mitigate this risk by confirming that each container exhibits the expected resource-usage and response time patterns and by providing the rationale for their implementation in section 4. The minimal reference service characterises background power consumption and possible temporal effects of the testbed. Comparisons with this control describe deviations from background conditions rather than the overhead that could be removed through refactoring.

Conclusion Validity. Measurement accuracy and statistical inference represent additional threats. DRAM power readings, collected via RAPL, reflect system-wide consumption and cannot be isolated at the container level, though we eliminate competing workloads to limit noise. Each experiment is repeated 30 times, warm-up periods are excluded, and HC3 robust standard errors are used to account for non-normality and heteroskedasticity in the residuals. Furthermore, Section 4 documents the concrete implementation choices and their rationale for each antipattern, enabling verification, replication, and adaptation of our implementations by other researchers.

11 Conclusion and Future Work

Software performance antipatterns are recurring situations in software design that are known to degrade system performance. Although these antipatterns are well recognized, limited evidence exists regarding their impact on software energy efficiency. This study investigates how response time relates to instantaneous power consumption and energy-related behavior. We conducted a controlled experiment using the ten performance antipatterns presented by [39]. Each antipattern is implemented as a containerized service, exercised under a fixed workload, and profiled to measure performance and power metrics. The results show that performance antipatterns do not exhibit uniform power behaviour, and these differences can have distinct implications for total energy consumption and energy efficiency. In some cases, performance issues that increase response time are also associated with increasing power demand and may imply higher energy use for the same amount of useful work, such as in The Ramp antipattern. The relationship between response time and power is also conditional on CPU utilization for some antipatterns. In Unnecessary Processing, for example, the average response-time coefficient is not significant, but the positive interaction indicates that its association with CPU power becomes more positive as CPU utilization increases. We encourage practitioners to consider response time, power consumption, and total energy as related but distinct metrics when evaluating software performance. As future work, we plan to inject these antipatterns into established microservice benchmarks (e.g., Train Ticket, TeaStore) and replicate the study at the system level and across different platforms. This will enable the identification of design patterns that promote better trade-offs between performance and energy efficiency.

Data Availability Statement

Experimental artifacts, datasets, antipattern implementations, and analysis scripts are available in the replication package [3]. The package includes a README file with instructions for reproducing the study.

References

- 1 Marcelo Amaral, Huamin Chen, Tatsuhiko Chiba, Rina Nakazawa, Sunyanan Choochotkaew, Eun Kyung Lee, and Tamar Eilam. Process-Based Efficient Power Level Exporter. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, 2024. doi:10.1109/CLoud62652.2024.00058.
- 2 T. W. Anderson and D. A. Darling. A Test of Goodness of Fit. *Journal of the American Statistical Association*, 49(268), 1954.
- 3 Alessandro Aneggi, Vincenzo Stoico, and Andrea Janes. Replication Package for the paper “Beyond Response Time: Investigating the Power Behavior of Performance Antipatterns”. <https://doi.org/10.5281/zenodo.21978998>, August 2026. doi:10.5281/zenodo.21978998.
- 4 José Miguel Aragón-Jurado, Abdul Ali Bangash, Bernabé Dorronsoro, Karim Ali, Abram Hindle, and Patricia Ruiz. Does faster mean greener? Runtime and energy trade-offs in iOS applications with compiler optimizations. *Sustainable Computing: Informatics and Systems*, 2025.
- 5 Gabriel Araújo, Vandırleya Barbosa, Luiz Nelson Lima, Arthur Sabino, Carlos Brito, Iure Fé, Paulo Rego, Eunmi Choi, Dugki Min, Tuan Anh Nguyen, et al. Energy consumption in microservices architectures: a systematic literature review. *IEEE Access*, 12, 2024. doi:10.1109/ACCESS.2024.3389064.
- 6 Alberto Avritzer, Ricardo Britto, Catia Trubiani, Barbara Russo, Andrea Janes, Matteo Camilli, André van Hoorn, Robert Heinrich, Martina Rapp, and Jörg Henß. A Multivariate Characterization and Detection of Software Performance Antipatterns. In *In Proc. ACM/SPEC Int. Conf. on Performance Engineering*. Association for Computing Machinery, 2021. doi:10.1145/3427921.3450246.
- 7 Alberto Avritzer, Andrea Janes, Catia Trubiani, Helena Rodrigues, Yuanfang Cai, Daniel Sadoc Menasché, and Álvaro José Abreu de Oliveira. Architecture and Performance Anti-patterns Correlation in Microservice Architectures. In *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. IEEE, 2025.
- 8 Alberto Avritzer, Daniel Menasché, Vilc Rufino, Barbara Russo, Andrea Janes, Vincenzo Ferme, André van Hoorn, and Henning Schulz. PPTAM: Production and Performance Testing Based Application Monitoring. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE ’19. Association for Computing Machinery, 2019. doi:10.1145/3302541.3311961.
- 9 Rabih Bashroush and Eoin Woods. Architectural Principles for Energy-Aware Internet-Scale Applications. *IEEE Software*, 34(3), 2017. doi:10.1109/MS.2017.60.
- 10 Victor Basili, Adam Trendowicz, Martin Kowalczyk, Jens Heidrich, Carolyn Seaman, Jürgen Münch, and Dieter Rombach. *Aligning Organizations Through Measurement: The GQM+Strategies Approach*. The Fraunhofer IESE Series on Software and Systems Engineering. Springer International Publishing, Cham, 1 edition, 2014. doi:10.1007/978-3-319-05047-8.
- 11 T. S. Breusch and A. R. Pagan. A Simple Test for Heteroscedasticity and Random Coefficient Variation. *Econometrica*, 47(5), 1979.
- 12 Simon Brown. *The C4 Model*. O’Reilly Media, 2026.
- 13 Vittorio Cortellessa, Antinisca Di Marco, and Catia Trubiani. Software Performance Antipatterns: Modeling and Analysis. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science*, pages 290–335. Springer, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-30982-3_9.

- 14 Vittorio Cortellessa, Daniele Di Pompeo, Vincenzo Stoico, and Michele Tucci. Software Model Refactoring Driven by Performance Antipattern Detection. *SIGMETRICS Perform. Eval. Rev.*, 49(4), 2022. doi:10.1145/3543146.3543159.
- 15 Vittorio Cortellessa, Daniele Di Pompeo, Vincenzo Stoico, and Michele Tucci. Many-objective optimization of non-functional attributes based on refactoring of software models. *Information and Software Technology*, 157, 2023. doi:10.1016/J.INFSOF.2023.107159.
- 16 Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020. doi:10.1109/CCGrid49817.2020.00–45.
- 17 Markus Funke, Patricia Lago, Esther Adenekan, Ivano Malavolta, and Ilja Heitlager. Experimental Evaluation of Energy Efficiency Tactics in Industry: Results and Lessons Learned. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 2024. doi:10.1109/ICSA59870.2024.00024.
- 18 Hoffmann Geerd-Dietger and Majuntke Verena. PowerLetrics: An Open-Source Framework for Power and Energy Metrics for Linux. In *2025 IEEE/ACM 9th International Workshop on Green and Sustainable Software (GREENS)*. IEEE Computer Society, 2025. doi:10.1109/GR EENS66463.2025.00009.
- 19 Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Perform. Eval. Rev.*, 40(3), 2012. doi:10.1145/2425248.2425252.
- 20 Hubblo. Scaphandre: Energy consumption metrology agent, 2020. URL: <https://github.com/hubblo-org/scaphandre>.
- 21 jpwhite3. northwind-SQLite3: SQLite3 version of the Microsoft Access 2000 Northwind sample database, 2022. URL: <https://github.com/jpwhite3/northwind-SQLite3>.
- 22 Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), 2018. doi:10.1145/3177754.
- 23 Artem Kruglov, Giancarlo Succi, and Gcinizwe Dlamini. System Energy Consumption Measurement. In Artem Kruglov and Giancarlo Succi, editors, *Developing Sustainable and Energy-Efficient Software Systems*, SpringerBriefs in Computer Science, pages 27–38. Springer International Publishing, Cham, 2023. doi:10.1007/978-3-031-11658-2_3.
- 24 James G MacKinnon and Halbert White. Some heteroskedasticity-consistent covariance matrix estimators with improved finite sample properties. *Journal of Econometrics*, 29(3), 1985.
- 25 Raghad Matar and Jasmin Jahić. An approach for evaluating the potential impact of anti-patterns on microservices performance. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2023.
- 26 Rodrigo Morales, Ruben Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. Anti-patterns and the energy efficiency of Android applications, 2016. arXiv:1610.05711.
- 27 Adel Nouredine. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *18th International Conference on Intelligent Environments (IE2022)*, Biarritz, France, 2022.
- 28 Adel Nouredine and Olivier Le Goaer. Investigating the Impact of Software Design Patterns on Energy Consumption. In *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, 2025. doi:10.1109/ICSA65012.2025.00024.
- 29 Object Management Group. Business Process Model and Notation (BPMN) Version 2.0.2. Technical Report formal/2013-01-03, Object Management Group (OMG), 2013. URL: <http://www.omg.org/spec/BPMN/2.0.2>.
- 30 Pallets Projects. Flask – Welcome to Flask. <https://flask.palletsprojects.com/en/stable/>, 2025. Accessed: 2025-12-08.

- 31 Priyavanshi Pathania, Nikhil Bamby, Rohit Mehra, Samarth Sikand, Vibhu Saujanya Sharma, Vikrant Kaulgud, Sanjay Podder, and Adam P. Burden. Calculating Software's Energy Use and Carbon Emissions: A Survey of the State of Art, Challenges, and the Way Ahead. In *Proceedings of the 2025 IEEE/ACM 9th International Workshop on Green and Sustainable Software*, GREENS '25. IEEE Press, 2025.
- 32 Bjorn Pijnacker, Brian Setz, and Vasilios Andrikopoulos. Container-level Energy Observability in Kubernetes Clusters, 2025. doi:10.48550/arXiv.2504.10702.
- 33 Riccardo Pincioli, Aldeida Aleti, and Catia Trubiani. Performance Modeling and Analysis of Design Patterns for Microservice Systems. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, 2023. doi:10.1109/ICSA56044.2023.00012.
- 34 Francisco Ponce, Stefano Forti, Jacopo Soldani, and Antonio Brogi. Enhancing Failure Resilience of Cloud-Edge Microservices: The FREEDA Approach. In Claus Pahl, Andrea Janes, Tomas Cerny, Valentina Lenarduzzi, and Matteo Esposito, editors, *Service-Oriented and Cloud Computing*, Cham, 2025. Springer Nature Switzerland.
- 35 Giuseppe Procaccianti, Héctor Fernández, and Patricia Lago. Empirical evaluation of two best practices for energy-efficient software development. *Journal of Systems and Software*, 117, 2016.
- 36 Linda Rising, editor. *The patterns handbook: Techniques, strategies, and applications*. Cambridge University Press, 1998.
- 37 Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1–2), 2010. doi:10.14778/1920841.1920902.
- 38 Bart Smaalders. Performance Anti-Patterns: Want your apps to run faster? Here's what not to do. *Queue*, 4(1), 2006. doi:10.1145/1117389.1117403.
- 39 Connie U. Smith and Lloyd G. Williams. New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In *28th International Computer Measurement Group Conference, December 8-13, 2002, Reno, Nevada, USA, Proceedings*. Computer Measurement Group, 2002. URL: http://www.cmg.org/?s2member_file_download=/proceedings/2002/2160.pdf.
- 40 Connie U Smith and Lloyd G Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, 2003.
- 41 The Linux Kernel Community. Power Capping Framework — The Linux Kernel Documentation, 2025. URL: <https://docs.kernel.org/power/powercap/powercap.html>.
- 42 Imara van Dinten, Pouria Derakhshanfar, Annibale Panichella, and Andy Zaidman. The slow and the furious? Performance antipattern detection in Cyber-Physical Systems. *Journal of Systems and Software*, 210, 2024. doi:10.1016/J.JSS.2023.111904.
- 43 Roberto Verdecchia, Giuseppe Procaccianti, Patricia Lago, et al. Empirical evaluation of the energy impact of refactoring code smells. In *5th International Conference on Information and Communication Technology for Sustainability. ICT4S2018*. EasyChair, 2018.
- 44 Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering (2024 edition)*. Springer, 2024.
- 45 Xingwen Xiao, Chushu Gao, and Justus Bogner. On the Effectiveness of Microservices Tactics and Patterns to Reduce Energy Consumption: An Experimental Study on Trade-Offs. In *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, 2025. doi:10.1109/ICSA65012.2025.00025.
- 46 Xiaolong Xu, Wanchun Dou, Xuyun Zhang, and Jinjun Chen. EnReal: An Energy-Aware Resource Allocation Method for Scientific Workflow Executions in Cloud Environment. *IEEE Transactions on Cloud Computing*, 4(2), 2016. doi:10.1109/TCC.2015.2453966.
- 47 Yiming Zhao, Tiziano De Matteis, and Justus Bogner. How Does Microservice Granularity Impact Energy Consumption and Performance? A Controlled Experiment. In *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*. IEEE, 2025.