Watts This Smell: A Comprehensive Taxonomy of Software Energy Smells

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

***Abstract*—As software proliferates across domains, its aggregate energy footprint has become a major concern. To reduce software's growing environmental footprint, developers need to identify and refactor *energy smells*: source code implementations, design choices, or programming practices that lead to inefficient use of computing resources. Existing catalogs of such smells are either domain-specific, limited to performance anti-patterns, lack fine-grained root cause classification, or remain unvalidated against measured energy data. In this paper, we present a comprehensive, language-agnostic, taxonomy of software energy smells. Through a systematic literature review of 60 papers and exhaustive snowballing, we coded 320 inefficiency patterns into 12 primary energy smells and 65 root causes mapped to the primary smells. To empirically validate this taxonomy, we profile over 21,000 functionally equivalent Python code pairs for energy, time, and memory, and classified the top 3000 pairs by energy difference using a multi-step LLM pipeline, mapping 55 of the 65 root causes to real code. The analysis reveals that 71% of samples exhibit multiple co-occurring smells, memory-related smells yield the highest per-fix energy savings, while power draw variation across patterns confirms that energy optimization cannot be reduced to performance optimization alone. Along with the taxonomy, we release the labeled dataset, including energy profiles and reasoning traces, to the community. Together, they provide a shared vocabulary, actionable refactoring guidelines, and an empirical foundation for energy smell detection, energy-efficient code generation, and green software engineering at large.**

***Index Terms*—Energy Smell, Software Anti-Patterns, Performance Anti-Patterns, Green Software, Taxonomy.**

I. Introduction

Kent Beck defined code smells as code structures that suggest, and sometimes scream for, the possibility of refactoring [1]. The existence of these smells reduces code quality and negatively impacts maintainability of a software system [2], [3]. Code smells are classified into different categories, types, and variants based on their occurrence granularity, artifact, and scope [4]. It includes smells in traditional code, such as implementation [1], [5], design [6], [7], architecture smells [2], [8], test smells [9], [10], and configuration smells [11].

Similar to traditional code smells that mainly target the maintainability attribute of software quality, smells also appear in code that affect other quality aspects, such as energy efficiency. Energy smells are defined as source code anti-patterns that use hardware resources inefficiently, leading to unnecessary energy consumption at runtime [12], [13]. By identifying such smells, analogous to how traditional code smells guide maintainability improvements, developers can refactor them to improve energy efficiency and overall software performance and quality.

A common misconception in software engineering is treating execution performance and energy consumption as interchangeable metrics. Although energy correlates with performance [14], [15], the two measure distinctly different phenomena. Performance primarily reflects execution time, *i.e.,* how long the CPU actively processes instructions to complete a task [16]. Energy consumption, in contrast, depends on how code interacts with underlying hardware states, power domains, and dynamic power management systems [17], [18]. Critically, faster execution does not always lead to lower energy consumption [19], [20]. Since energy equals power multiplied by time ($E = P \times T$), and power draw varies across code patterns [19], a code change that reduces execution time can simultaneously increase power draw, resulting in higher total energy. This effect is well-documented in concurrent programming, where parallelism trades time for increased resource utilization [21]–[23]. Consequently, systems tuned exclusively for speed can harbor energy inefficiencies invisible to conventional profilers.

Beyond this conceptual distinction, the global software ecosystem is rapidly expanding. This growth is fueled by cloud computing, mobile platforms, microservices, and IoT deployments, and is expected to continue due to the rise of AI-generated software [24]. As software scales in size and deployment volume, its overall energy footprint increases accordingly. Even small implementation-level inefficiencies, when replicated across millions of running instances and billions of invocations, translate into significant energy waste [25], [26]. While energy inefficiency ultimately manifests at the level of hardware micro-architecture and power management, it is driven by high-level software decisions: algorithmic choices, data structure selection, API usage patterns, and implementation idioms that developers make long before any hardware is involved [27]–[29]. These factors together motivate the systematic study of energy smells in software engineering. Just as code smells equip the community with a shared vocabulary of structural anti-patterns signaling poor design, which subsequently operationalized into static analysis tools, refactoring catalogs, and quality metrics [1], [4], a well-defined taxonomy of energy smells can empower developers with actionable guidance for energy-aware implementation.

Despite existing catalogs of performance anti-patterns [30]–[35] and energy anti-patterns [12], [13], [36], several critical

gaps persist across this body of work. First, most performance-focused catalogs [30]–[35] define their taxonomies in terms of time efficiency, omitting energy consumption as a distinct optimization target. Second, existing taxonomies have a limited scope and do not comprehensively cover different types of smells. Dargan *et al.* [33] and Tao *et al.* [34], derived from student submissions, cover narrow problem domains and omit implementation-level concerns such as concurrent programming and hardware locality behavior. Third, several catalogs are domain- or language-specific by design, limiting their applicability. For example, the catalog proposed by Gottschalk *et al.* [13] covers Android-specific energy bugs derived from a literature survey. Fourth, validation of the catalog or energy profiling is either missing in current studies [13], [37] or conducted within a narrowly scoped setup [12].

To address these limitations, we build on existing energy and performance smell catalogs, identify a set of known issues leading to energy inefficiency, filter and consolidate them, and develop a comprehensive, language-agnostic, two-level taxonomy of software energy smells. The first level defines general energy smell categories, and the second level pinpoints the actionable and precise root causes. To ensure exhaustive coverage, we conduct a systematic literature review supplemented by forward and backward snowballing, encompassing 60 articles across diverse domains that identify issues leading to performance or energy inefficiencies. Following a multi-phase filtering and annotation process, two independent annotators apply open coding, axial coding, and selective coding to identify 320 relevant issues (Cohen's $\kappa = 0.74$). This process produces a consolidated **taxonomy of twelve energy smell categories and 65 root causes**. To empirically validate this taxonomy, we construct a profiled Python dataset based on `Pie-Perf` [38], containing pairs of functionally equivalent code snippets with measured energy, time, and memory consumption. We apply a multi-step classification pipeline using a state-of-the-art reasoning LLM (`DeepSeek-V3.2` [39]) to map the root causes of inefficiency in $3,000$ code pairs to the taxonomy, successfully identifying all twelve categories across 85% of our identified root causes, validating the real-world prevalence and relevance of our taxonomy.

In summary, this paper makes the following primary contributions:

- **A comprehensive taxonomy** of twelve energy smell categories and 65 root causes, derived from a systematic literature review. The taxonomy is language-agnostic, covering a variety of smells from low-level hardware effects to high-level algorithmic choices.
- **A labeled dataset** of $21,428$ code pairs with verified energy, memory, and time measurements, including a manually validated subset of $3,000$ pairs annotated with multi-label smell classifications and step-by-step reasoning traces to support future research.
- **An empirical analysis** of the prevalence, co-occurrence, and energy impact of each smell category, providing evidence that energy optimization cannot be reduced to performance optimization alone.

All artifacts, including the taxonomy, dataset, and annotations, are publicly available in our replication package [40].

II. Methodology

A. Taxonomy Design Goals

Energy smells *are source code implementations, design choices, or programming practices that lead to inefficient use of computing resources* [12], [13]. The primary **objective** of this research is to define a comprehensive, well-detailed, and language-agnostic taxonomy of software energy smells. To ensure broad applicability across diverse programming paradigms and application domains, the taxonomy must systematically categorize inefficiencies ranging from implementation-level issues to high-level algorithmic flaws. To achieve this, we structure the taxonomy into a two-level hierarchy:

Energy smell (Category): A high-level, observable issue in source code that uses computing resources inefficiently.

Root cause (Subcategory): The specific, actionable non-optimal or sub-optimal implementation choice—such as a poorly chosen data structure or a redundant loop—that triggers the broader energy waste.

A robust taxonomy requires that an inefficient implementation found in any application maps seamlessly to our defined categories and sub-categories. While strict mutual exclusivity is not a hard constraint—since programming logic often intertwines and one smell may correlate with or trigger another—we aim to design the taxonomy to minimize overlap. Each energy smell and its underlying root cause needs to maintain a clear boundary. This precise separation ensures that developers can pinpoint the exact source of an inefficiency, facilitating straightforward refactoring and classification.

To systematically capture and classify these inefficiencies, we perform a rigorous, multi-phase literature review and qualitative coding process, following established software engineering systematic mapping guidelines [41]. Fig. 1 illustrates the full pipeline.

B. Phase 1: Primary Literature Search

We initiate the literature search and collection process of relevant articles by querying Scopus, illustrated in the Step ❶ of Fig. 1. Scopus is a comprehensive database widely utilized in software engineering for literature reviews due to its extensive coverage and advanced search capabilities [42], [43]. We construct a targeted query designed to identify articles discussing code-level efficiency anti-patterns: *TITLE-ABS-KEY ((“energy” OR “power consumption” OR “green” OR “performance inefficiency”) AND (“code smell” OR “smell” OR “anti-pattern”) AND (“software” OR “code”)) AND (LIMIT-TO (SUBJAREA, “COMP”))*

We explicitly include “performance inefficiency” because software performance and energy consumption are highly correlated metrics [14], [15]. While strictly defined energy smells often relate to how code mismanages physical hardware components (*e.g.,* preventing CPU sleep states or triggering excessive memory fetches) [17], [18], total energy consumption

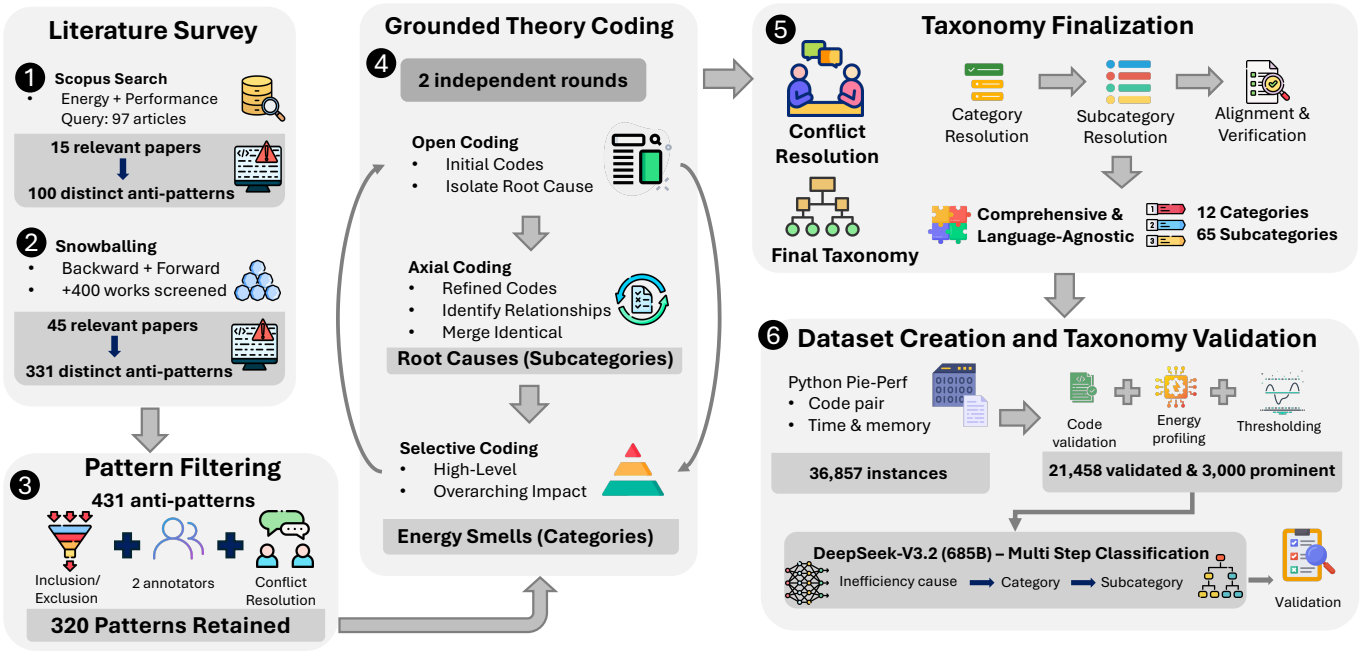

Fig. 1: Overview of the methodology

is fundamentally governed by execution time as expressed by $E = P \times T$. Also, the literature detailing performance anti-patterns can contribute significantly to our taxonomy, because these issues are often less application-specific [33], [35] than pure energy studies [12], [13]. Hence, including these issues will help meet our goal of broader applicability. Nonetheless, we evaluate performance findings cautiously to ensure they legitimately impact energy consumption.

The initial Scopus search yields 97 relevant articles. We apply strict inclusion and exclusion criteria to filter this set. We *include* any study that identifies, defines, or empirically proves a transferable performance or energy inefficiency at the code level, regardless of its original domain (*e.g.,* extracting language-agnostic practices from infrastructure-specific studies [44]). We also include studies discussing specific logic and implementation issues, from implementation-level to high algorithmic-level. We *exclude* studies on design smells, *e.g.,* feature envy, that primarily target maintainability rather than energy efficiency directly. We also exclude duplicate reports, vague definitions, and overly niche, application-specific implementations that cannot transfer to other domains. Notably, about 29% of the captured articles focus on Android, 24% examine general software inefficiencies or the energy impact of refactoring code smells and design choices, and 11% target domains such as `web`, `cloud`, `IoT`, and `SQL`. The remainder address other software smells (e.g., test smells or code smell detection), with a small portion being irrelevant to our search. After this filtering, we retain 15 relevant studies, extracting 100 distinct performance and energy anti-patterns along with their definitions.

C. Phase 2: Snowballing

To maximize coverage beyond the initial Scopus results, we use the 15 retained articles as seed papers and conduct comprehensive snowballing. Then, we capture all code snippets in papers retrieved from snowballing reported as being energy- or performance-inefficient, as depicted in Step ❷ of Fig. 1.

We begin with backward snowballing on all papers identified through the Scopus search. For each newly discovered paper, we apply the same extraction procedure and recursively examine its references and related work sections. In addition, we perform forward snowballing on highly relevant and foundational studies. In particular, performance anti-pattern studies by Smith *et al.* [30]–[32] are extensively expanded forward, as subsequent papers citing them frequently propose additional relevant issues.

Additionally, we include grey literature referenced within primary or snowballed sources when it reports validated inefficiencies. For example, we include the `CAST` Software's 979 green rules, derived from accepted standards such as PCI [45], MITRE's CWE [46], and OMG's CISQ [47]. Overall, we screen more than 400 works to identify studies reporting performance or energy anti-patterns.

After filtering out studies that merely repeat previously identified issues and that comply with our inclusion and exclusion criteria, we retain a total of 45 studies, among them 19 targeting specifically energy-efficiency and 26 performance. Combined with the 15 Scopus papers, this yields a final corpus of 60 relevant studies. From these, we extract **331 distinct issues**, bringing the total identified potential energy-efficiency issues to **431** (100 from Scopus + 331 from snowballing).

D. Phase 3: Relevance Filtering

In Step ❸ of Fig. 1, we classify each of the 431 collected issues as *relevant* or *irrelevant*. The primary objective is to retain language-agnostic constructs that directly impact energy consumption, time complexity, or space complexity. To ensure an unbiased process, the first two authors, each with more than a year of experience in software energy consumption research, independently label all the extracted items based on the predefined inclusion and exclusion criteria. Out of 431 total items, 43 result in disagreement between the two annotators, yielding a Cohen's κ of 0.74, indicating substantial inter-annotator agreement. The annotators then met to discuss and resolve all conflicts in accordance with the taxonomy goals of comprehensiveness and applicability across languages and domains. During this resolution, we exclude instances that are overly vague, purely design-level with no direct performance impact, or exclusively tied to specific applications or languages (*e.g.,* mobile-specific "`Faulty GPS Behavior`" or Java-specific "`Use Static Final for Constants`"). We retain original context-specific implementations if their underlying inefficiency generalizes across domains. For example, database-specific "`Unnecessary Row Retrieval`" generalizes to fetching unnecessary data in any program. Furthermore, we also retain traditional code smells that inherently introduce resource inefficiencies, as well as low-level hardware and memory behaviors (*e.g.,* "Inefficient Array Declaration Order"). This ensures our taxonomy comprehensively spans the full spectrum of inefficiencies. At the end of this phase, 320 items and their definitions are retained.

E. Phase 4: Grounded Theory Coding Process

To systematically analyze the 320 retained items towards taxonomy generation, we adopt a grounded theory coding approach proposed by Strauss and Corbin [48], as recommended for qualitative analysis in software engineering research [49]–[51]. As shown in Step ❹ of Fig. 1, both annotators perform this process independently across two complete rounds, applying three iterative coding steps:

1) **Open coding**: A line-by-line analysis of each extracted item to isolate its fundamental "root cause", stripping away application-specific contexts to identify the underlying language-agnostic mechanism. We assign an initial code (*e.g.,*"`Fetching never used data`" or "`Using a suboptimal memory-heavy data structure`") to each instance that describes the technical behavior causing the inefficiency.
2) **Axial coding**: The initial codes are compared against one another to identify relationships, redundancies, and shared technical contexts. Codes that describe functionally identical root causes under different names are merged, while complex codes that conflate multiple distinct mechanisms are split.
3) **Selective coding**: The refined codes are abstracted into high-level categories. We examine the conceptual boundaries of the subcategories and group them based on their overarching impact on energy consumption, system resources, and execution flow. For example, codes dealing with unused outputs, repeated identical calculations, and dead branches are grouped under the encompassing category of *Redundant Computation*.

To ensure clarity and immediate readability, we adopt a bipartite naming convention documented in traditional code smell literature [4], [7]. Each name consists of two components: (1) a *prefix adjective* serving as an inefficiency modifier (*e.g., Redundant*, *Unnecessary*, *Suboptimal*, *Missing*) that defines how energy is wasted, and (2) a *suffix noun* identifying the specific programmatic construct or behavior involved (*e.g., Computation*, *Data Structures*, *Control Flow*).

F. Phase 5: Taxonomy Finalization

In the final phase, *i.e.,* Step ❺ of Fig. 1, both the annotators collaboratively finalize the taxonomy. This step is conducted jointly and follows a strict top-down resolution order to ensure minimal overlap and maximum comprehensiveness.

a) Category resolution: We first reconcile the high-level categories proposed by both annotators to establish the final set of energy smells.

- **Clear one-to-one mapping (category–category):** If both annotators propose categories with similar explanations, we merge them, combine their definitions, and assign a unified name following the naming convention.
- **Clear one-to-one mapping (category–sub-category):** If one annotator defines an issue as a category and the other as a sub-category, we jointly evaluate its scope. Issues reflecting a general theme are finalized as categories; issues representing a specific root cause are retained as subcategories.
- **No clear mapping:** For categories proposed by only one annotator, we assess granularity. General issues are added as new categories; highly granular issues that qualify as root cause are retained as subcategories. If, after discussion, an issue does not fit either level, it is discarded.

After this step, all categories are agreed upon, and the taxonomy contains the final set of energy smells, while root causes remain to be mapped.

b) Sub-category resolution: Next, we align the combined list of sub-categories from both annotators.

- **Clear one-to-one mapping:** Subcategories with a clear correspondence are merged. We apply the naming convention, consolidate definitions into a precise explanation, and create an explicit "example" column derived from grounded literature. This example reflects the subcategory's behavior and supports validation. Each confirmed subcategory is then assigned to the most appropriate fixed category.
- **No clear mapping:** Remaining subcategories are evaluated collaboratively to avoid overlap. Extremely granular codes that apply only in narrow situations are absorbed into an existing subcategory as specific instances. Codes

with appropriate granularity and no overlap with existing subcategories are formalized as new subcategories and assigned to the most fitting category.

c) Alignment verification: After all sub-categories are placed, we perform a final validation pass. We verify that each subcategory, along with its definition, resides under the most appropriate category. If a subcategory shows stronger alignment with a different category based on its definition similarity, it is reassigned.

This finalization process produces a comprehensive, language-agnostic taxonomy consisting of 12 categories (*i.e.,* energy smells) and 65 sub-categories (*i.e.,* root causes).

G. Dataset Creation and Taxonomy Validation

We construct an annotated dataset of energy smells to empirically validate our proposed taxonomy. This process is demonstrated in Step ❻ of Fig. 1. Given a pair of functionally equivalent code snippets with distinct energy profiles, an underlying energy smell must account for the difference. Mapping the exact root cause of the inefficiency in these code pairs to our taxonomy confirms the real-world validity of our categories and sub-categories. This mapping also reveals the prevalence of specific energy smells, identifies subcategories that may not appear in practice, and surfaces potential gaps if an inefficiency cannot be mapped.

1) Data selection and filtering: We utilize the Python splits of the Pie-Perf dataset [38], which is derived from IBM CodeNet [52]. Each instance in this dataset contains a problem description, a pair of efficient and inefficient code snippets, and their respective CPU time and memory consumption. We select Pie-Perf because it provides a highly diverse, application-agnostic collection of implementations that effectively captures a wide range of real-world developer habits and algorithmic approaches. To ensure data integrity, we first verify that both code versions in a pair are functionally correct and yield the same output. Out of $36,857$ initial instances, $21,428$ pairs successfully pass the functional test cases. These pairs form our filtered base dataset.

2) Energy profiling and metric extraction: Pie-Perf inherently focuses only on execution time and memory; therefore, we must empirically measure the energy consumption of each pair. Following the best practices for accurate energy profiling of software [53], we execute a rigorous energy measurement pipeline. For each of the $21,428$ functionally correct pairs, we randomly select 50 of the 100 available test cases. To eliminate hardware state noise, we perform a warm-up run followed by three measured iterations for both the efficient and inefficient snippets. We use the Linux *perf* command-line tool to capture system-wide CPU and RAM energy consumption alongside elapsed execution time. Concurrently, we utilize *time* tool to record the maximum resident set size (RSS), capturing peak physical memory allocation during execution. This process appends six verified metrics (energy, memory, and time for both snippet versions) to each instance.

3) Energy-based sample selection: To focus on pairs with meaningful energy differences, we sort the $21,428$ pairs by the absolute energy difference between their efficient and inefficient versions. We observe that energy differences become negligible beyond the $3,000^{th}$ sample; thus, we select the top $3,000$ pairs as our final sample. This threshold provides sufficient coverage to validate all smell categories while ensuring that observed energy differences can be confidently attributed to the identified smells rather than noise or external factors.

4) Multi-step LLM classification pipeline: Mapping $3,000$ complex algorithmic pairs to specific energy smells requires advanced reasoning, making exhaustive manual classification impractical. Consequently, we delegate the classification task to `DeepSeek-V3.2 (685B)` [39], a frontier model that demonstrates state-of-the-art reasoning capabilities. We configure the model with a temperature of 0 for deterministic output, allocate $4,096$ maximum reasoning tokens per step, disable caching, and process each instance independently.

To mitigate priming bias [54] and prevent the model from hallucinating fits to our taxonomy, we divide the classification into a three-step sequential pipeline by designing prompts that operate as follows:

1) **Root cause analysis.** Given the problem description and few-shot examples, the model compares functionally equivalent Python implementations across energy, time, and memory metrics to identify concrete code-level inefficiencies. Crucially, taxonomy labels are *not* provided to ensure unbiased reasoning derived strictly from the code.
2) **Category triage.** The model performs a multi-label semantic match, mapping the unbiased root causes extracted in Step 1 to one or more high-level energy smell categories based on provided definitions.
3) **Subcategory classification.** Using provided definitions and examples, the model refines Step 2's candidates into precise, multi-label subcategories. To ensure actionable results, only independent, refactoring-relevant root causes are reported: downstream symptoms that disappear when fixing a primary smell are ignored, whereas multiple independent inefficiencies are all captured.

The dataset, along with the reasoning trace output of all the steps, is made available with the dataset [40].

5) Dataset evaluation: To evaluate our classification pipeline, we select a random subset of 100 samples, and the first two authors independently validate each sample. In Step 1, the model accurately identified the root cause of inefficiency in all 100 cases without exposure to our taxonomy. Reviewing these explanations revealed no novel inefficiencies outside our defined categories, providing empirical validation for the taxonomy's comprehensiveness. Because Step 2 is an intermediate mapping phase, we focused our remaining evaluation on Step 3, which is the final classification step, to ensure the pipeline accurately isolated primary root causes rather than downstream effects. The annotators achieved substantial initial agreement (Cohen's $\kappa = 0.65$). The annotators then discussed and mutually resolved all conflicting labels to form a final ground-truth dataset. Comparing this resolved ground-truth against the pipeline's output yielded an overall accuracy of 94%. The 6 misclassifications involved highly complex,

multifaceted algorithmic flaws in which isolating a single root cause is inherently ambiguous, confirming that the error rate remains minimal and acceptable.

III. Energy Smell Taxonomy

Table I presents the full taxonomy of twelve energy smells and 65 root causes, where each energy smell tells a developer *what kind* of inefficiency is present and each root cause tells them *what to detect and how to fix it*. We empirically validate the taxonomy by classifying 3,000 code pairs from the *Pie-Perf* dataset using the three-step LLM pipeline (Sections II-G, II-G4). In this section, we introduce the twelve categories organized into four thematic groups, explain category boundary decisions, demonstrate classification on real code, and report empirical coverage.

a) Wasted work (C1, C2, C4): Redundant Computation (C1) covers code whose results are never consumed or that repeats computation already done: dead code, redundant assignments, repeated expressions, and unnecessary initializations. *Unnecessary Call Overhead* (C2) targets function calls, method delegation, and dynamic dispatch that add cost without proportional benefits. *Inefficient Control Flow* (C4) addresses branching logic that performs unnecessary checks, uses suboptimal evaluation order, or prevents runtime optimization.

b) Iteration and loop structure (C3): Inefficient Iteration Patterns (C3) captures loops that do more per-iteration work than necessary, iterate more times than required, or fail to terminate early. Due to their repetitive nature, issues in this category often amplifies issues from other categories (Section III-C).

c) Data and memory (C5, C6): Suboptimal Data Structures (C5) covers choosing containers that do not match the workload's access pattern, such as using a list for frequent membership queries when a set provides constant-time lookup. *Unnecessary Memory Usage* (C6) captures allocating, copying, or retaining memory beyond what the workload needs, including unnecessary object creation, premature materialization of lazy sequences, and over-allocation.

d) Algorithm and reuse (C7, C8): Suboptimal Algorithmic (C7) covers choosing an algorithm or decomposition strategy with higher complexity than the problem requires. *Missing Reuse* (C8) addresses failing to store and reuse results of expensive computations across calls, including missing memoization and redundant data fetching.

e) External systems and concurrency (C9, C11): Inefficient External Data Access (C9) targets inefficient interaction with databases, file systems, and APIs. *Inefficient Concurrency Management* (C11) captures thread and task misuse: excessive lock contention, serial bottlenecks, and missed parallelism opportunities.

f) Language and hardware (C10, C12): Underused Language Primitives (C10) covers not using available optimized built-in functions or runtime constructs. *Poor Hardware Locality* (C12) targets cache misses, branch misprediction, and suboptimal memory access patterns relevant when code interacts with contiguous-memory libraries or native extensions.

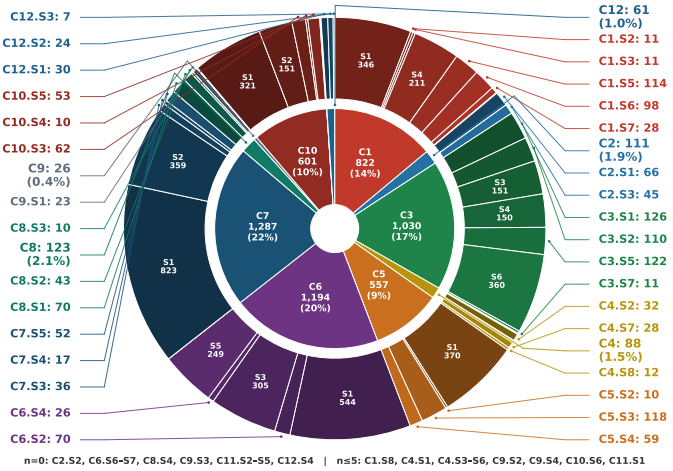

Fig. 2: Energy smells and root causes distribution

A. Empirical Coverage

As detailed in Section II-G4, applying our classification pipeline to the Pie-Perf dataset yielded 3,000 successfully categorized code pairs. We observe 55 (84.6%) out of the 65 subcategories at least once in the 3,000 classified samples. Fig. 2 depicts the prevalence of all subcategories. The frequency distribution reflects strong algorithmic stress-testing in the dataset, dominated by *Suboptimal Algorithm Choice* (C7.S1, 822) and *Inefficient Problem Decomposition* (C7.S2, 359). The high occurrence of *Unnecessary Object Creation* (C6.S1, 543) and *Inefficient Data Structure Choice* (C5.S1, 370) suggests developers often prioritize quick logical correctness over memory-efficient abstractions. Similarly, frequent *Skipping of Optimized Built-ins* (C10.S1, 321) indicates a tendency to reimplement logic instead of using native library functions, reflecting limited language-specific performance awareness. Finally, the prevalence of *Unfiltered Bulk Iteration* (C3.S6, 360) and *Unnecessary Data Materialization* (C6.S3, 304) reveals a bias toward eager execution rather than energy-efficient lazy evaluation or generator pipelines.

The absent subcategories can be grouped in three clusters: concurrency patterns (4 out of 5 C11 subcategories), requiring multi-threaded execution absent from single-threaded competitive code; external data access (C8.S4, C9.S3), requiring database or I/O interaction; and resource-lifecycle patterns (C6.S6, C6.S7), requiring persistent connections or multi-call objects. C2.S2 and C12.S4 are also absent due to Python's dispatch model and C-extension specificity, respectively. Every absent subcategory maps to a structural feature absent from the dataset's domain, not to a gap in the taxonomy.

B. Taxonomy Characterization

The root causes are not uniformly distributed across categories. At the structural level, C1, C4 (eight root causes each), and C3 (7) are the largest, reflecting the density of distinct mechanisms at the implementation level, while C2 (3) and C5 (4) are smaller because they address fewer, more structurally distinct mechanisms. The empirical prevalence from the 3,000

TABLE I: Energy smell taxonomy: 12 energy smells and 65 root causes.

Category	Root Cause (ID)	Description
C1: Redundant Computation	Dead Code (C1.S1)	Code with **unused results** or **unreachable paths**.
	Redundant Assignment (C1.S2)	Statements leaving **state unchanged** (self-assignment, overwriting unread values).
	Redundant Control Flow (C1.S3)	Control flow that **does not alter execution** regardless of branch taken.
	Repeated Computation (C1.S4)	**Identical calculations repeated** in the same scope without data changes.
	Resultless Computation (C1.S5)	Operations producing **no subsequently used output**.
	Unnecessary Initialization (C1.S6)	Setup on paths where results are **unused or overwritten**.
	Unnecessary Variable (C1.S7)	Intermediate variables whose **overhead exceeds reuse** utility.
	Excessive Runtime-Mgmt (C1.S8)	**Excessive housekeeping** or logging invocations wasting energy.
C2: Unnecessary Call Overhead	Unnecessary Delegation (C2.S1)	**Wrappers or accessors** adding overhead without benefit.
	Missing Static Declaration (C2.S2)	**Instance methods** for operations independent of instance state.
	Excessive Modularization (C2.S3)	Hot paths **over-split** into tiny layers where call overhead rivals useful work.
C3: Inefficient Iteration Patterns	Inefficient Iteration Construct (C3.S1)	Iteration forms with **avoidable overhead** when efficient alternatives exist.
	Recomputing Loop-Invariant (C3.S2)	**Loop-invariant values** recomputed inside loops instead of once outside.
	Inefficient Per-Iteration Setup (C3.S3)	**Heavy initialization** inside loops that could be pre-computed.
	Inefficient Nested Iteration (C3.S4)	**Nested loops** where an equivalent $O(n)$ approach exists.
	Missing Loop Early Exit (C3.S5)	Failing to **exit early** after determining the required result.
	Unfiltered Bulk Iteration (C3.S6)	Processing **entire collections** when only a subset is needed.
	Inefficient Array Mutation (C3.S7)	**Modifying collections during iteration**, causing re-indexing or anomalies.
C4: Inefficient Control Flow	Poor Short-Circuit Ordering (C4.S1)	**Poor operand ordering** preventing optimal short-circuit evaluation.
	Redundant Conditional (C4.S2)	Evaluating **statically known** or **unchanged** conditions.
	Inefficient Conditional Nesting (C4.S3)	**Deep nesting** hindering early returns and branch optimization.
	Missing Else-If (C4.S4)	**Independent conditionals** for mutually exclusive branches.
	Expensive Comparison (C4.S5)	Costly **reflection or type introspection** instead of simpler comparisons.
	Expensive Exception Flow (C4.S6)	**Exceptions for expected control flow**, incurring high object creation costs.
	Missing Edge-Case Guards (C4.S7)	Missing **early guards** for trivial inputs, forcing expensive general-case processing.
	Non-Idiomatic Condition (C4.S8)	**Non-idiomatic** conditional patterns missing runtime optimizations.
C5: Suboptimal Data Structures	Inefficient Structure Choice (C5.S1)	Data structures with **suboptimal complexity** for the dominant operation.
	Missing Helper Type (C5.S2)	Missing **auxiliary structures** (caches, indices) to optimize repeated lookups.
	Over-Provisioned Data Type (C5.S3)	**Heavier types** than data constraints necessitate.
	Unnecessary Representation (C5.S4)	**Format conversions** without net efficiency gains.
C6: Unnecessary Memory Usage	Unnecessary Object (C6.S1)	**Excessive, short-lived, or duplicate** objects increasing GC pressure.
	Unnecessary Copying (C6.S2)	**Copying** when views, references, or in-place operations suffice.
	Unnecessary Materialization (C6.S3)	Fully **materializing** intermediate data instead of lazy evaluation.
	Oversized Data Retaining (C6.S4)	Retaining **full structures** when smaller representations (IDs) suffice.
	Over-Allocation (C6.S5)	**Over-allocating** buffers, wasting memory and increasing cache pressure.
	Leaked Resource Handles (C6.S6)	Failing to **release external resources** (files, sockets, connections).
	Leaking Mutable Defaults (C6.S7)	**Mutable default parameters** causing silent persistence and growth.
C7: Suboptimal Algorithmic	Suboptimal Algorithm Choice (C7.S1)	Algorithms with **worse complexity** when efficient alternatives exist.
	Inefficient Decomposition (C7.S2)	**Redundant or poorly-ordered** subcomputations.
	Avoidable Recursion (C7.S3)	**Recursion** where iterative solution is more efficient.
	Inefficient Operation Ordering (C7.S4)	**Expensive operations first** when cheaper ones could prune input.
	Unsimplified Operation (C7.S5)	Heavy operations on inputs that could be **pre-reduced or simplified**.
C8: Missing Reuse	Missing Memoization Cache (C8.S1)	**Recomputing** deterministic results instead of memoizing.
	Missing Derived-Value Reuse (C8.S2)	**Recreating** expensive artifacts (regex, SQL) on every use.
	Missing Local Lookup Caching (C8.S3)	Repeating **global/attribute lookups** in hot code instead of caching locally.
	Redundant Data Fetching (C8.S4)	**Repeatedly fetching** unchanged external data instead of caching.
C9: Inefficient External Access	Fragmented I/O Calls (C9.S1)	**Unbatched** small I/O or network operations.
	Oversized Data Retrieval (C9.S2)	Fetching **more data than consumed** by application logic.
	Inefficient Retrieval Paths (C9.S3)	**Multiple retrieval steps** instead of a single joined query.
	Expensive Query Pattern (C9.S4)	Query structures **inherently costly** for database engines.
C10: Underused Language Primitives	Skipping Optimized Built-ins (C10.S1)	**Manual reimplementation** of optimized built-in functionality.
	Inefficient Built-in Choice (C10.S2)	Using constructs when **faster equivalents** exist.
	Missing Bulk Primitive Usage (C10.S3)	**Per-element operations** instead of vectorized/batched primitives.
	Inefficient String Concat (C10.S4)	**Repeated concatenation** of immutable strings, causing excessive allocation.
	Inefficient Scope Lookup (C10.S5)	Variables in scopes with **unnecessary lookup overhead**.
	Expensive Operator Overloads (C10.S6)	**Expensive overloaded methods** invoked via operator syntax on non-primitives.
C11: Inefficient Concurrency	Excessive Lock Contention (C11.S1)	**Overly broad synchronization** serializing parallelizable work.
	Forced Serial Bottleneck (C11.S2)	**Single-file execution** through shared resource, negating parallelism.
	Leaked Background Threads (C11.S3)	**Leaked threads/tasks** without clean termination.
	Blocking The Main Thread (C11.S4)	**Blocking main thread** with long-running CPU or I/O work.
	Missed Parallelism (C11.S5)	**Serial execution** of independent work, underutilizing resources.
C12: Poor Hardware Locality	Inefficient Large-Stride (C12.S1)	**Large stride** array traversal causing frequent cache misses.
	Sparse Element Access (C12.S2)	**Scattered access** in large structures, preventing efficient cache use.
	Unpredictable Branches (C12.S3)	**Data-dependent branching** in tight loops, defeating branch prediction.
	Inefficient Array Declaration (C12.S4)	**Hot buffers after cold ones**, resulting in poor cache alignment.

classified samples, however, reveals a different ranking. C7 (only five root causes) is the most prevalent category at 41.6% of samples, followed by C6 at 38.9% and C3 at 33.3%. At the other end, C11 and C9 each appear in fewer than 1% of samples, which is expected given the dataset's single-threaded, in-memory domain.

The taxonomy spans three abstraction levels. *Implementation-level* categories (C1, C2, C3, C4, C6, C10) target small code snippets fixable by editing a few lines. *Design-level* categories (C5, C7, C8) require choosing a different data structure, algorithm, or caching strategy. *Architecture-level* categories (C9, C11) involve interaction with external systems or concurrency models; C12 is syntactically implementation-level but operates at the hardware level. The empirical data confirms this distinction: design-level smells produce higher median energy savings per fix ($1,537$ J, N=$1,725$) than implementation-level smells ($1,146$ J, N=$2,476$), consistent with the expectation that deeper design fixes yield larger gains. Architecture-level (N=28) and hardware-level (N=61) smells show lower medians (415 J and 561 J), though this partly reflects their under-representation in competitive programming code.

C. Category Design and Boundary Decisions

Designing the taxonomy required resolving cases where the same inefficiency could plausibly belong to more than one category. The guiding principles are to minimize overlap so that each inefficiency maps to exactly one primary root cause, and to ensure that the category and subcategory boundaries support distinct refactoring actions. Below we explain the non-obvious boundary decisions. Where applicable, we supplement these with disambiguation evidence from the classification pipeline, in which a sample initially matched multiple candidate categories but was resolved to one based on the taxonomy rules.

a) Redundant vs. Repeated computation (C1 vs. C8): Both categories involve redundant work, but they differ in scope. C1 covers computations whose result has no impact on the final functional outcome: dead code, self-assignments, and unused initializations. The fix is deletion, since the work never needed to happen. C8 covers computation that is necessary and produces a useful result, but the program recomputes it from scratch on every call instead of caching it. The fix is not deletion but adding a reuse mechanism, such as memoization or precomputation. This distinction carries over to tooling: detecting C1 requires only intra-procedural analysis to identify unused results, while detecting C8 requires tracking repeated calls with identical inputs across invocations.

b) Loops as amplifiers (C3 vs. other categories): Loops multiply any inefficiency by the iteration count, which is why several C3 root causes have counterparts elsewhere. *Per-Iteration Setup* (C3.S3) captures waste from creating objects *inside a loop body* that should be hoisted out; *Missing Derived-Value Reuse* (C8.S2) captures the same artifact being recreated *across function calls*. Similarly, a hand-written summation loop could be classified as C3.S1 or C10.S1; we assign C10.S1 because the root cause is not using the available built-in function.

c) Data structures vs. algorithms (C5 vs. C7): C5 addresses choosing the wrong *container* for the workload's access pattern; C7 addresses choosing the wrong *algorithm* or decomposition strategy. The two require different refactoring actions even when they co-occur.

D. Overlap and Classification Rules

A single code fragment can trigger multiple energy smells simultaneously. In the classified dataset, 71% of the $3,000$ samples received two or more labels (48% received exactly two, 20.4% received three), confirming that multi-label classification is the norm, not an edge case. Samples with multiple labels exhibit significantly higher energy waste: multi-label cases have a median energy savings of $1,659$ J compared to 549 J for single-label cases (Mann-Whitney U, $p < 0.0001$), a $3.0\times$ difference. This indicates that energy waste compounds when multiple inefficiencies interact.

To handle this overlap during classification, we established a strict decision rule: *assign the root cause whose fix directly eliminates the primary inefficiency, rather than tagging downstream consequences or amplifiers.* Out of the $3,000$ samples, 765 (25.5%) initially had multiple candidate categories in Step 2 but were resolved to a single, deeper root cause using this rule.

E. Classification Walkthrough

We present two examples from the dataset to illustrate how the taxonomy classifies real code.

a) Example 1: Unused import as dead code (Dataset #4327): The inefficient version contains `import numpy as np` but never references NumPy. Importing this library is costly because it loads large C-extensions and other initialization, which consumes CPU and memory before any application logic runs. Removing this single line saves $4,036$ J ($13.5\times$ ratio). The LLM considered both C1 (dead code) and C6 (memory overhead from loading NumPy's C extensions). The final classification is C1.S1 (Dead Code): the root cause is that the import should not exist; the memory overhead is a consequence, not an independent problem.

b) Example 2: Loop mutation amplifying a suboptimal algorithm (Dataset #14434): A Sieve of Eratosthenes implementation calls `list.pop()` inside a while loop, producing $O(n^2)$ complexity from repeated element shifting. The efficient version uses iterative primality testing, saving $23,446$ J ($67\times$ ratio). This sample receives two labels: C3.S7 (array mutation) and C7.S1 (suboptimal algorithm). Neither identification is complete independently: the algorithm choice is the root cause, but the in-loop mutation is the mechanism that makes it quadratic. Notably, C3.S7 appears only 11 times but has the highest mean savings ($9,000$ J), showing that rare smells can be costly when present.

IV. Discussion

This section examines the empirical findings from the $3,000$ classified code pairs and discusses their implications for practitioners and researchers.

a) Frequency and impact do not align: The distribution of energy savings in our dataset is highly right-skewed, exhibiting a mean of $2,561$ J compared to a much lower median of $1,081$ J (a $2.4\times$ ratio). With a skewness of 3.4 and an extreme range (84 J to $38,416$ J), the data demonstrates a clear Pareto-like effect: a small fraction of severe inefficiencies accounts for a disproportionate majority of the total energy waste. Crucially, the frequency of an energy smell does not dictate its impact. For example, C7 is the most common category (41.6%) but its median savings ($1,119$ J) rank only third. In contrast, C5 (18.0%) and C6 (38.9%) yield the highest median savings at $3,989$ J and $3,670$ J, respectively. This impact is magnified in the extreme upper tail: when isolating the top 25% most wasteful code pairs (Q4, savings above $3,996$ J), C6 and C5 dominate, appearing in 57.9% and 29.2% of samples (χ^2, $p < 0.001$). Minor issues like C4 and C2 are nearly absent from this high-impact tier. This mismatch arises because C5 and C6 heavily affect the memory subsystem. Large unnecessary allocations increase DRAM traffic, elevate garbage collection pressure, and raise power draw. While an algorithmic smell primarily extends execution time, a memory smell extends both time and power, producing exponentially higher savings per fix. The practical implication for software tooling is clear: **energy smell detectors must prioritize warnings by category impact rather than mere frequency.** A single C5 or C6 finding is worth substantially more energy than ten C4 findings (median ≈ 307 J).

b) Co-occurrence reveals C6 as a structural hub: The five most common co-occurring pairs are C3+C7 (426 samples), C1+C6 (372), C6+C7 (341), C5+C6 (303), and C3+C6 (295). C6 appears in four of the top five and co-occurs significantly with 8 of 11 other categories (Fisher's exact test, $p < 0.05$). Bad algorithms over-allocate proportionally to wasted work; redundant computations create persisting temporary objects; wrong data structures are themselves over-allocation; inefficient iteration forces intermediate collections.

c) Energy is not a proxy for time: Three layers of evidence support an energy-specific taxonomy beyond performance taxonomies. First, power draw is not constant: it ranges from 60.5 W to 147.3 W (Coefficient of Variation (CV) of 24.2%) for the inefficient code. Memory-intensive categories draw significantly more power: C5 has a mean power ratio of 1.267 and C6 of 1.186, compared to 1.056 for C7 and 1.035 for C8. Fixing a data structure smell reduces both time and power, a double benefit invisible to time-only analysis. Second, 15 samples show improved time with increased energy, all involving NumPy vectorization: faster wall time but SIMD activation at higher power (*e.g.,* 78 W to 135 W), producing up to 64% energy increase. Conversely, 5 samples show improved energy with worse time, driven by power reduction alone. These 20 cases confirm that time and energy improvements are distinct axes. Third, in 69.1% of samples, the energy ratio (E_{v0}/E_{v1}) exceeds the time ratio (T_{v0}/T_{v1}), meaning naive estimation using $E = \text{constant} \times T$ systematically underestimates energy waste because inefficient code tends to draw more power in addition to running longer.

A. Implications

As Green Software Engineering matures, a critical gap remains between measuring energy waste and actively refactoring code to eliminate it. Our two-level taxonomy bridges this gap by formalizing the path from measurement to remediation. By establishing a shared vocabulary and an empirical foundation, this work streamlines the efforts towards the development of automated tools and integrates energy awareness directly into standard workflows.

Our taxonomy equips **software developers** with a universal language for energy efficiency, where the high-level *energy smell* serves as a diagnostic signal, while the *root cause* provides an immediately actionable refactoring path that optimize resource usage, execution speed, and client battery life. For **industry**, organizations can integrate energy smell detection directly into CI/CD pipelines as a proactive safeguard against wasteful code, allowing companies to quantify software quality, benchmark products against industry standards, and significantly reduce server electricity costs while positioning themselves as environmentally responsible. Furthermore, **AI model developers** can utilize our profiled dataset as a foundation for downstream tools, enabling the supervised training of lightweight, real-time IDE linters and using verified energy, time, and memory metrics, along with the reasoning traces for preference-based alignment (e.g., DPO [55]) to teach generative LLMs to synthesize energy-efficient code. Finally, this work opens numerous avenues for **researchers** to conduct large-scale empirical studies, tracking longitudinal software evolution to investigate prevalence of energy smells, and develop novel metric to quantify them.

V. Related Work

Software energy concern in general: Software energy consumption has long been a research focus. Early work by Mehta *et al.* [56] explores compiler-level optimizations—such as improved register allocation, loop unrolling, software pipelining, and recursion elimination—to enhance energy efficiency. Similarly, Sinha *et al.* [57] develop software energy models, showing that restructuring algorithms can significantly improve energy-quality scalability in resource-constrained systems. Recent studies highlight software's role in managing hardware power states [27], [28]. In fact, Georgiou *et al.* [27] advocate incorporation of energy efficient practices in the embedded software systems. Noureddine *et al.* [58] introduce a fine-grained runtime framework for estimating Java energy use to detect software energy hotspots, and Georgiou *et al.* [29] survey techniques, emphasizing best practices and continuous monitoring for improved energy efficiency of software.

Impact of software design on energy: A distinct line of studies analyzes the effect of software design—such as

design patterns [59], code smells [60], and refactoring techniques [61]—on software energy consumption. Recent systematic literature reviews [42], [62] comprehensively explore this domain, identifying exactly which code smells, refactoring methods, and design patterns positively or negatively impact energy consumption.

Energy smells: Although software design influences energy use, it mainly targets maintainability. We further divide the studies focusing on energy smells in two broad categories.

1) Energy anti-pattern definition: Brandolese *et al.* [36] study source-level C transformations for reducing power, identifying anti-patterns such as *oversized loop bodies* and *inefficient array access*. Similarly, Hopfner *et al.* [63] introduce a resource substitution matrix for networking, identifying anti-patterns such as *transmitting uncompressed data* or *neglecting local caching*. Many studies in this domain focus specifically on Android applications, as energy directly affects both the user experience and the environment [60], [64]. Gottschalk *et al.* [13] identify the first set of Android energy code smells through a literature review, cataloging patterns such as *binding resources too early, loop bug, immortality bug* alongside their detection and refactoring methods. Similarly, Li *et al.* [65] demonstrate that bundling network packets and optimizing array length reads effectively lower mobile energy consumption. Additionally, energy efficiency is also critical for battery-powered embedded systems with limited resources [66]. Vetro *et al.* [12] present the first validated catalog of energy smells for embedded systems, empirically comparing smelly and refactored code. Their catalog includes anti-patterns such as *parameter by value and dead local store*.

In broader domains, GreenForLoops [37] examines Java energy efficiency, identifying loop-related smells such as *array initialization*, *string concatenation*, and *unnecessary nested loops*. Pereira *et al.* [67] and Oliveira *et al.* [68] analyze the energy impact of alternative Java data structures, showing benefits of optimal substitutions. In the web domain, Rani *et al.* [69] port Android energy anti-patterns to web applications, demonstrating that *reduce size*, *cache*, and *batch operations* improve efficiency.

2) Energy smells detection: Several tools target energy inefficiencies, predominantly in Android. EcoAndroid [70] automates energy-efficient Java refactorings based on existing literature. Prestat *et al.* [71] evaluate static-analysis tools such as PAPRIKA [72] and ADOCTOR [73] for detecting mobile resource issues, while ecoCode [74] broadens prior work by defining novel Android energy smells and providing a dedicated detection tool.

Performance anti-patterns: Given the relationship between performance and energy, examining performance-specific inefficiencies is crucial. Lyu *et al.* [75] define SQL performance anti-patterns, demonstrating how these database-level issues degrade both runtime and energy on mobile devices. Exploring general software, Zhao *et al.* [35] analyze 192 real-life performance issues in Java projects, utilizing grounded theory to categorize bottlenecks into eight root causes (*e.g., inefficient data structures*) and map them to optimization solutions. At the statement level, Tao *et al.* [34] profile 250 student assignments to extract 27 recurring inefficiencies. Similarly, Dargan *et al.* [33] analyze slow Python submissions, deriving a taxonomy that groups over 750 efficiency bugs into three primary root causes (*superfluous computation, suboptimal data structures*, and *suboptimal algorithms*) spanning twelve specific subcategories.

VI. Threats to Validity

Construct validity. The LLM classification pipeline may introduce misclassification or bias. We mitigate this by keeping the first step independent of the taxonomy to capture genuine root causes, validating on 100 random instances (94% accuracy), and relying on a large sample size (3,000) for statistical robustness. Energy measurement on a single hardware configuration is another construct threat; we follow established best practices [53] to minimize noise, though absolute joule values may vary across architectures.

Internal validity. Confining the initial search to Scopus risks missing papers with non-standard terminology. We mitigate this through multiple query iterations and exhaustive forward and backward snowballing, expanding the corpus from 15 to 60 resources. The taxonomy may also include patterns with marginal energy impact; we mitigate this by extracting patterns from peer-reviewed, highly cited sources and established standards. While domain-specific implementations may be absent, specific bad practices typically function as instances of existing subcategories, and the framework is extensible to accommodate new root causes as they emerge.

External validity. The Pie-Perf dataset focuses on single-threaded, algorithmic competitive programming, explaining the low prevalence of C9 (External Data) and C11 (Concurrency). We mitigate this by deriving the taxonomy from cross-domain literature; the dataset validates 84.6% of subcategories, with absent entries mapping to missing domain features rather than taxonomic gaps. Empirical validation uses only Python, so prevalence and energy magnitudes reflect CPython behavior. We mitigate this by keeping all definitions language-agnostic, abstracting root causes from Python-specific syntax so the underlying inefficiencies generalize to other high-level languages.

VII. Conclusions and Future Work

This study addresses the lack of standardization in defining, reporting, and resolving software energy inefficiencies. We conduct a systematic literature review and apply grounded theory coding to develop a language-agnostic, two-level taxonomy of twelve high-level energy smells and 65 actionable root causes. We then empirically validate the framework on profiled Python code pairs using a multi-step LLM classification pipeline. Similar to traditional code smells, our taxonomy establishes a shared vocabulary and practical guidelines for sustainable software development.

Building on this taxonomy and dataset, we plan to validate its cross-language applicability and evaluate smell prevalence

across diverse domains. We will also mine real-world open-source repositories (*e.g.,* GitHub) to measure how often these root causes occur and are refactored in practice. Finally, we aim to develop a lightweight smell detector integrated in popular IDEs for real-time detection and train generative AI models to automatically identify, explain, and refactor energy-inefficient code.

REFERENCES

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[2] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *International conference on the quality of software architectures*. Springer, 2009.

[3] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, 2011.

[4] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, 2018.

[5] W. H. Brown, R. C. Malveau, H. W. S. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[6] D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, and J. Wegener, "Dependence anti patterns," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*. IEEE, 2008.

[7] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[8] T. Sharma, P. Singh, and D. Spinellis, "An empirical investigation on the relationship between design and architecture smells," *Empirical Software Engineering*, 2020.

[9] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 2001.

[10] M. Greiler, A. Van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013.

[11] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th international conference on mining software repositories*, 2016.

[12] A. Vetro, L. Ardito, G. Procaccianti, M. Morisio *et al.*, "Definition, implementation and validation of energy code smells: an exploratory study on an embedded system," in *Proceedings of ENERGY 2013: The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, 2013.

[13] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter, "Removing energy code smells with reengineering services," in *INFORMATIK 2012*. Gesellschaft für Informatik eV, 2012.

[14] J. Pallister, S. J. Hollis, and J. Bennett, "Identifying compiler options to minimize energy consumption for embedded platforms," *The Computer Journal*, 2015.

[15] K. Chan-Jong-Chu, T. Islam, M. M. Exposito, S. Sheombar, C. Valladares, O. Philippot, E. M. Grua, and I. Malavolta, "Investigating the correlation between performance scores and energy consumption of mobile web apps," in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, 2020.

[16] D. A. Patterson and J. L. Hennessy, *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[17] Z. Li, R. Grosu, P. Sehgal, S. A. Smolka, S. D. Stoller, and E. Zadok, "On the energy consumption and performance of systems software," in *Proceedings of the 4th Annual International Conference on Systems and Storage*, 2011.

[18] A. Carroll and G. Heiser, "Unifying dvfs and offlining in mobile multicores," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014.

[19] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy efficiency across programming languages: how do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, 2017.

[20] M. Weber, C. Kaltenecker, F. Sattler, S. Apel, and N. Siegmund, "Twins or false friends? a study on energy consumption and performance of configurable software," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023.

[21] S. Abdulsalam, Z. Zong, Q. Gu, and M. Qiu, "Using the greenup, powerup, and speedup metrics to evaluate software energy efficiency," in *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2015.

[22] R. Zambre, L. Bergstrom, L. A. Beni, and A. Chandramowlishwaran, "Parallel performance-energy predictive modeling of browsers: Case study of servo," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 2016.

[23] B. Goetz, *Java concurrency in practice*. Pearson Education, 2006.

[24] K. Z. Cui, M. Demirer, S. Jaffe, L. Musolff, S. Peng, and T. Salz, "The effects of generative ai on high-skilled work: Evidence from three field experiments with software developers," *Management Science*, 2026.

[25] L. Lannelongue, J. Grealey, and M. Inouye, "Green algorithms: quantifying the carbon footprint of computation," *Advanced science*, 2021.

[26] A. S. Andrae and T. Edler, "On global electricity usage of communication technology: trends to 2030," *Challenges*, 2015.

[27] K. Georgiou, S. Xavier-de Souza, and K. Eder, "The iot energy challenge: A software perspective," *IEEE Embedded Systems Letters*, 2017.

[28] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "On reducing the energy consumption of software: From hurdles to requirements," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020.

[29] S. Georgiou, S. Rizou, and D. Spinellis, "Software development lifecycle for energy efficiency: techniques and tools," *ACM Computing Surveys (CSUR)*, 2019.

[30] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Proceedings of the 2nd international workshop on Software and performance*, 2000.

[31] C. U. Smith and L. Williams, "New software performance antipatterns: More ways to shoot yourself in the foot," in *Int. CMG Conference*, 2002.

[32] C. Smith and L. G. Williams, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Computer Measurement Group Conference*, 2003.

[33] H. Dargan, A. Gilbert-Diamond, A. J. Hartz, and R. C. Miller, "" why is my code slow?" efficiency bugs in student code," in *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, 2025.

[34] Y. Tao, W. Chen, Q. Ye, and Y. Zhao, "Beyond functional correctness: An exploratory study on the time efficiency of programming assignments," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*, 2024.

[35] Y. Zhao, L. Xiao, X. Wang, L. Sun, B. Chen, Y. Liu, and A. B. Bondi, "How are performance issues caused and resolved?-an empirical study from a design perspective," in *Proceedings of the ACM/SPEC international conference on performance engineering*, 2020.

[36] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "The impact of source code transformations on software power and energy consumption," *Journal of Circuits, Systems, and Computers*, 2002.

[37] R. P. Gurung, J. Porras, and J. Koistinaho, "Static code analysis for reducing energy code smells in different loop types: a case study in java," in *2024 10th International Conference on ICT for Sustainability (ICT4S)*. IEEE, 2024.

[38] A. Madaan, A. Shypula, U. Alon, M. Hashemi, P. Ranganathan, Y. Yang, G. Neubig, and A. Yazdanbakhsh, "Learning performance-improving code edits," *arXiv preprint arXiv:2302.07867*, 2023.

[39] DeepSeek-AI, "Deepseek-v3.2: Pushing the frontier of open large language models," 2025.

[40] anonymous, "Watts this smell: A comprehensive taxonomy of software energy smells," Mar. 2026. [Online]. Available: https://doi.org/10.5281/zenodo.18896365

[41] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and software technology*, 2015.

[42] O. Poy, M. Á. Moraga, F. García, and C. Calero, "Impact on energy consumption of design patterns, code smells and refactoring techniques: A systematic mapping study," *Journal of Systems and Software*, 2025.

[43] V. Garousi and M. V. Mäntylä, "Citations, research topics and active countries in software engineering: A bibliometrics study," *Computer Science Review*, 2016.

[44] S. Kosbar and M. Hamdaqa, "Smells-sus: Sustainability smells in iac," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 2025.

[45] PCI Security Standards Council, "Payment Card Industry Data Security Standard (PCI DSS)," 2026. [Online]. Available: https://www.pcisecuritystandards.org

[46] The MITRE Corporation, "Common Weakness Enumeration (CWE)," 2026. [Online]. Available: https://cwe.mitre.org

[47] Consortium for Information & Software Quality (CISQ), "Automated Source Code Quality Measures (ISO/IEC 5055)," 2026. [Online]. Available: https://www.it-cisq.org

[48] A. Strauss and J. Corbin, "Basics of qualitative research techniques," 1998.

[49] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, 2006.

[50] S. Jahan, S. S. Rajput, T. Sharma, and M. M. Rahman, "Why attention fails: A taxonomy of faults in attention-based neural networks," *arXiv preprint arXiv:2508.04925*, 2025.

[51] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 international symposium on empirical software engineering and measurement*. IEEE, 2011.

[52] IBM CodeNet, "IBM Project CodeNet," 2026. [Online]. Available: https://github.com/IBM/Project_CodeNet

[53] S. Rajput, T. Widmayer, Z. Shang, M. Kechagia, F. Sarro, and T. Sharma, "Enhancing energy-awareness in deep learning through fine-grained energy measurement," *ACM Transactions on Software Engineering and Methodology*, 2024.

[54] N. Chen, J. Liu, X. Dong, Q. Liu, T. Sakai, and X.-M. Wu, "Ai can be cognitively biased: An exploratory study on threshold priming in llm-based batch relevance assessment," in *Proceedings of the 2024 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region*, 2024.

[55] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in neural information processing systems*, 2023.

[56] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh, "Techniques for low energy software," in *Proceedings of the 1997 international symposium on Low power electronics and design*, 1997.

[57] A. Sinha and A. P. Chandrakasan, "Energy aware software," in *VLSI Design 2000. Wireless and Digital Imaging in the Millennium. Proceedings of 13th International Conference on VLSI Design*. IEEE, 2000.

[58] A. Noureddine, R. Rouvoy, and L. Seinturier, "Monitoring energy hotspots in software: Energy profiling of software code," *Automated Software Engineering*, 2015.

[59] S. Maleki, C. Fu, A. Banotra, and Z. Zong, "Understanding the impact of object oriented programming and design patterns on energy efficiency," in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2017.

[60] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," *Information and Software Technology*, 2019.

[61] G. Pinto, F. Soares-Neto, and F. Castor, "Refactoring for energy efficiency: A reflection on the state of the art," in *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. IEEE, 2015.

[62] D. Connolly Bree and M. Ó Cinnéide, "How software design affects energy performance: A systematic literature review," *Journal of Software: Evolution and Process*, 2025.

[63] H. Höpfner and C. Bunse, "Towards an energy-consumption based complexity classification for resource substitution strategies." in *Grundlagen von Datenbanken*, 2010.

[64] A. Carette, M. A. Ait Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017.

[65] D. Li and W. G. Halfond, "An investigation into energy-saving programming practices for android smartphone app development," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, 2014.

[66] C. Guo, S. Ci, Y. Zhou, and Y. Yang, "A survey of energy consumption measurement in embedded systems," *IEEE Access*, 2021.

[67] R. Pereira, M. Couto, J. Saraiva, J. Cunha, and J. P. Fernandes, "The influence of the java collection framework on overall energy consumption," in *Proceedings of the 5th International Workshop on Green and Sustainable Software*, 2016.

[68] W. Oliveira, R. Oliveira, F. Castor, B. Fernandes, and G. Pinto, "Recommending energy-efficient java collections," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019.

[69] P. Rani, J. Zellweger, V. Kousadianos, L. Cruz, T. Kehrer, and A. Bacchelli, "Energy patterns for web: An exploratory study," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*, 2024.

[70] A. Ribeiro, J. F. Ferreira, and A. Mendes, "Ecoandroid: An android studio plugin for developing energy-efficient java mobile applications," in *2021 IEEE 21st international conference on software quality, reliability and security (QRS)*. IEEE, 2021.

[71] D. Prestat, N. Moha, and R. Villemaire, "An empirical study of android behavioural code smells detection," *Empirical Software Engineering*, 2022.

[72] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien, "Detecting antipatterns in android apps," in *2015 2nd ACM international conference on mobile software engineering and systems*. IEEE, 2015.

[73] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of android-specific code smells: The adoctor project," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017.

[74] O. Le Goaer and J. Hertout, "Ecocode: A sonarqube plugin to remove energy smells from android projects," in *Proceedings of the 37th IEEE/ACM International conference on automated software engineering*, 2022.

[75] Y. Lyu, A. Alotaibi, and W. G. Halfond, "Quantifying the performance impact of sql antipatterns on mobile applications," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019.