IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Green Refactoring For Energy-Efficient Java Software: A Literature Review

Sumit Kumar and Dr. Jasvir Singh

Department of Computer Science and Engineering, Punjabi University, Patiala, Punjab, India

Abstract—Software energy use is now a measurable, growing slice of global electricity demand, and much of that footprint comes down to how code is written, not what hardware it runs on. Green refactoring, restructuring existing source code to cut its energy footprint without changing what it does, has been proposed as a practical way to close that gap without waiting on new hardware or new languages. This paper reviews the literature on green refactoring for Java, a language that still runs most enterprise and Android backend systems even though sustainability research has largely moved on to mobile and cloud-native platforms. We searched five major databases, applied a structured inclusion and exclusion protocol, and screened the results down to a final set of primary studies covering refactoring techniques, profiling tools, and their reported energy effects. The review sorts green refactoring into six technique categories, compares five widely used Java profiling tools, and lays out what the empirical literature agrees and disagrees on. The clearest finding is that results are inconsistent: energy savings from the same technique vary by workload, JVM version, and measurement tool, and few studies isolate cause from effect cleanly. We identify six open gaps, among them the lack of multi-metric evaluation standards and the near-total absence of automated, profiling-guided refactoring tools for Java, and close with six directions for future work, including AI-assisted refactoring and energy-aware practices for microservice architectures.

Keywords—*green software engineering, energy efficiency, Java, refactoring, JVM, runtime profiling, literature review, sustainable computing*

I. INTRODUCTION

Data centers, mobile devices, and everyday consumer software now use enough electricity that the industry can no longer treat it as a rounding error. Most of the conversation about cutting that footprint has centered on hardware, cooling, and renewable sourcing. Far less attention goes to the code itself. Two programs that produce identical output can differ in energy draw by an order of magnitude. Memory allocation, data structure choice, and how heavily a program leans on the runtime underneath it all play a part. A benchmark that ran the same algorithms across dozens of programming languages found energy differences wide enough that language choice alone counts as a sustainability decision [2]. If language choice alone can shift energy use that much, smaller, code-level decisions inside a single language should matter too.

Green software engineering (GSE) grew directly out of that observation. It treats energy and resource use as a first-class quality attribute, on the same level as correctness, performance, and maintainability. Green refactoring is GSE's maintenance-side branch. It uses the same mechanics as ordinary refactoring, restructuring code without changing what it does, just pointed at a different target: instead of improving readability or cutting coupling, the goal is measurable energy reduction. One refactoring catalog frames this explicitly as a refactoring problem rather than a purely architectural one, arguing that many long-standing code smells carry an energy cost on top of their usual design cost [26].

Java is a reasonable place to study this, and an overlooked one. It still powers a large share of enterprise backend systems and remains the dominant language for Android. But it runs on a managed runtime, the JVM, and that runtime adds behavior, garbage collection, JIT compilation, dynamic class loading, that compiled languages simply don't have. That layer changes how refactoring choices translate into energy outcomes. One study removed textbook code smells from a sample of applications and found the energy savings were far from uniform: some refactorings helped, some did nothing, and a few made things measurably worse [3]. Vincent, a JIT-level optimization, was built specifically to cut the energy cost of "hot methods" identified at runtime, a research direction that only makes sense because the JVM's own execution model is part of the problem [8].

Even so, the literature on green refactoring is scattered. Some studies focus on code smells, some on runtime profiling tools, some on design patterns, and very few connect all three. A systematic mapping study covering the intersection of design patterns, code smells, and refactoring techniques found substantial disagreement on effect direction and size across primary studies [25]. A broader systematic literature review on how software design choices affect energy performance, published in 2025, covers design decisions generally rather than refactoring specifically, and does not isolate Java from other managed and unmanaged languages [4]. Neither review is wrong to take a broad view, but that breadth leaves a gap: nobody has yet produced a Java-specific, non-Android-specific synthesis that brings refactoring techniques, the JVM behavior shaping their effects, and the profiling tools used to measure them into one place. Filling that gap is this review's main contribution.

A. Research Objectives

This review is organized around four objectives:

1) Identify and classify the green refactoring techniques that have been proposed or evaluated for Java software.

2) Compare the runtime profiling tools used in the empirical literature to measure the energy and resource effects of these techniques.

3) Synthesize what the existing empirical evidence agrees and disagrees on regarding the energy impact of specific refactoring categories.

4) Identify open research gaps and propose concrete directions for future work on energy-efficient Java software.

B. Major Contributions

This paper makes the following contributions:

1) A structured, PRISMA-guided review of green refactoring for Java, built from a documented search and screening protocol rather than an informal survey.

2) A six-category taxonomy of green refactoring techniques for Java, each grounded in specific primary studies rather than generic best-practice advice.

3) A consolidated comparison of five runtime profiling tools used in Java energy research, merging overlapping tool-comparison tables from prior work into a single reference table.

4) A cross-study synthesis that names where the literature agrees, where it disagrees, and why. It doesn't just list findings side by side.

5) Six concrete research gaps paired with six corresponding future research directions, each tied to specific unresolved questions in the current body of work.

6) A deliberately narrow scope: Java broadly, not Android specifically. Two recent 2025–2026 reviews cover software design generally or the Android ecosystem specifically; neither addresses the general-purpose Java landscape in between, and this review fills that gap.

II. RESEARCH METHODOLOGY

This review follows the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) protocol, adapted for software engineering literature. The goal is to keep the search, screening, and extraction process auditable instead of ad hoc.

A. Research Questions

RQ1: What green refactoring techniques have been proposed or evaluated for Java software, and how are they categorized in the literature?

RQ2: What runtime profiling tools and metrics are used to measure the energy impact of these techniques?

RQ3: What does the empirical evidence say about the actual energy effects of different refactoring categories, and where does it disagree?

RQ4: What gaps remain in current research, and what directions would close them?

B. Search Strategy

Search strings combined terms for green software engineering, energy efficiency, refactoring, and Java. These ran across five databases: IEEE Xplore, ACM Digital Library, Springer Link, ScienceDirect, and arXiv, the last used for recent preprints not yet indexed elsewhere.The nine search strings used were:

1) "green refactoring" AND Java

2) "energy efficient" AND refactoring AND Java

3) "code smells" AND energy AND Java

4) "software energy consumption" AND JVM

5) "runtime profiling" AND energy AND Java

6) "green software engineering" AND refactoring

7) "energy efficiency" AND "design patterns" AND Java

8) "sustainable software" AND Java AND refactoring

9) "power consumption" AND "Java Virtual Machine"

Searches were restricted to studies published between 2015 and 2026, to capture both the foundational refactoring-energy literature and the most recent tool and taxonomy work.

C. Inclusion Criteria

Studies were included if they: (a) addressed energy or power consumption at the code, method, or component level in Java; (b) evaluated a refactoring technique, code smell, or design pattern with a reported or measurable energy effect; (c) used or described a runtime profiling or benchmarking methodology; and (d) were peer-reviewed conference papers, journal articles, or, where directly relevant, arXiv preprints and theses with a clearly documented empirical protocol. Studies that used an established benchmark suite as their evaluation basis, the Renaissance suite for JVM workloads, for instance, were counted as meeting a baseline methodological standard for inclusion. A shared benchmark simply makes cross-study comparison more meaningful than ad hoc micro-benchmarks do [16].

D. Exclusion Criteria

Studies were excluded if they: (a) addressed energy efficiency purely at the hardware or data center infrastructure level with no code-level component; (b) focused exclusively on Android UI or battery-life optimization without a generalizable code-refactoring angle; (c) lacked any empirical measurement, relying only on theoretical or opinion-based claims; or (d) were duplicate publications of the same study across venues, in which case the more complete version was retained.

E. Selection Process

The initial search across the five databases returned a combined pool of records. These went through de-duplication, then title-and-abstract screening, then a full-text check against the inclusion and exclusion criteria above. Fig. 1 summarizes the flow.

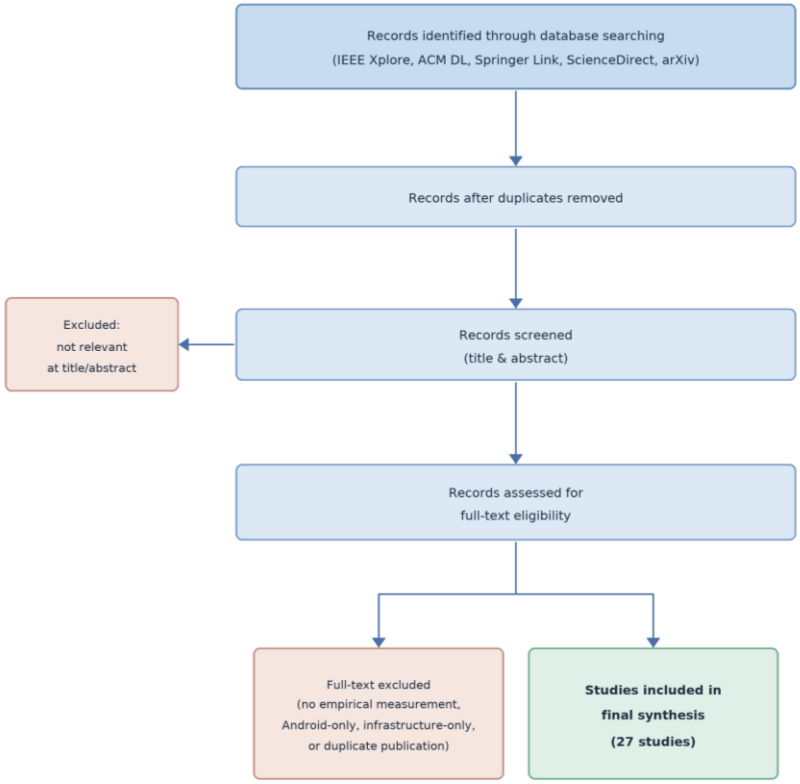

Fig. 1. Study selection process (PRISMA-style flow).

F. Data Extraction

For every included study, a consistent set of fields was extracted to support later synthesis, listed in Table I.

TABLE I. DATA EXTRACTION FIELDS

Field	Description
Study identifier	Author(s), year, venue
Refactoring technique(s)	Which category of technique was studied (see Section IV)
Programming language(s)	Java only, or Java compared against other languages
Energy metric(s) used	Joules, watts, CPU energy, package energy, etc.
Profiling tool(s) used	Tool name and measurement layer (hardware, OS, JVM, application)
Benchmark/workload	Real application, synthetic benchmark, or benchmark suite
Reported effect direction	Energy reduction, increase, mixed, or no significant effect
Effect magnitude	Percentage or absolute change where reported
Threats to validity noted	Author-acknowledged limitations

G. Quality Assessment

Each study was rated on three criteria. Was the measurement methodology described clearly enough to reproduce? Was the sample size, meaning the number of applications, methods, or runs, large enough to support its conclusions? And were confounding factors such as JIT warm-up, background processes, and JVM version differences addressed? Studies that were weak on all three were kept only where they added something unique to the taxonomy or tool coverage, and that limitation is flagged wherever it matters in Sections IV through VI.

H. Data Synthesis

Findings were synthesized narratively rather than through meta-analysis, since the primary studies use different metrics, different workloads, and different reporting units, and pooling them quantitatively would not be statistically sound. Instead, results are grouped by refactoring category and profiling

approach, with explicit attention to where studies agree or disagree on effect direction.

III. GREEN SOFTWARE ENGINEERING AND ENERGY BEHAVIOR IN JAVA

A. Evolution of Green Software Engineering

Green software engineering did not start out as a single, defined discipline. It grew out of adjacent fields, energy-aware hardware design, sustainable computing, and later empirical software engineering, as researchers began asking whether the rigor applied to performance testing could carry over to energy testing. A tertiary study of the sustainability literature traces this accumulation and notes that measurement standardization has lagged well behind conceptual interest in the topic [27]. That lag matters; it is one of the tensions this review keeps coming back to. The Green Software Measurement Model (GSMM) addresses part of the problem directly, giving researchers and practitioners a shared vocabulary and measurement approach for resource and energy efficiency instead of leaving each study to invent its own metric [23]. A wider strand of work argues that environmentally sustainable computing needs to be treated holistically, across the full software and infrastructure stack, rather than as a code-level afterthought, though that scope sits above the refactoring-level detail this review focuses on [24]. This line of thinking has also been extended into a model-based approach, where sustainability analysis and improvement get folded directly into software modeling activities rather than bolted on after implementation [10]. Together, these works suggest green software engineering has matured conceptually faster than it has methodologically, which is exactly the gap a focused, Java-specific technical review can usefully fill.

B. Energy Behavior in the Java Virtual Machine

To understand why refactoring changes energy consumption in Java, you have to understand what the JVM is actually doing underneath the source code. This subsection walks through those mechanisms. Section V covers the tools used to observe them in practice; Section VI covers what researchers actually measured when they did. Fig. 2 summarizes the chain of effect, from a refactoring decision in source code through to the energy figure a profiler eventually reports.

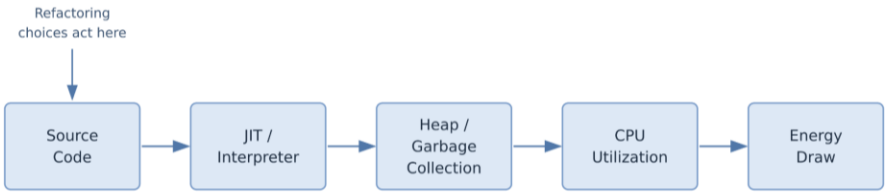

Fig. 2. Mechanism chain linking source-level refactoring to measured energy draw.

1) CPU Utilization. CPU cycles are the most direct proxy for energy draw, since modern processors scale power use with load. In Java, CPU utilization depends less on the source code's algorithmic complexity alone than on how the JIT compiler treats a given method: one that runs rarely stays interpreted, while one that runs often gets compiled to native code, which changes its CPU profile mid-execution. So the same refactored method can look completely different energy-wise depending on how often it runs, not just how it's written.

2) Heap Behavior. The JVM heap is where most Java-specific energy cost hides. Even short-lived, temporary objects cost CPU cycles to allocate and later to collect. Refactoring that increases object churn, replacing a mutable, reusable buffer with a stream of small immutable objects for readability's sake, say, can quietly push energy cost up even as it makes the code cleaner.

3) Garbage Collection. From an energy standpoint, garbage collection is not background housekeeping. It is an active CPU consumer competing directly with application logic. Different GC algorithms (G1, ZGC, Parallel GC) trade off collection frequency against pause time, and refactoring that changes an application's allocation rate shifts which GC behavior dominates. A refactoring that looks neutral in wall-clock time can

still push a meaningful share of total CPU time from application code over to garbage collection.

4) Allocation Patterns. Beyond raw object count, allocation size and lifetime matter too. Short-lived objects that die young are cheap for generational collectors to handle, while objects that survive into older generations are expensive to promote and eventually collect. Decisions around caching, object pooling, and which collection type to use directly determine which of those categories an object falls into.

5) Execution Time. Execution time and energy consumption correlate, but they are not the same thing. One study on Java method energy usage makes this point directly, arguing that static code metrics alone are poor predictors of method-level energy use and that execution time needs to be measured alongside static structure to draw reliable conclusions [17]. A method can run faster while burning more energy per unit time if it shifts work onto power-hungrier code paths, which is why energy needs its own measurement rather than standing in for timing.

6) Runtime Energy Hotspots. Put these mechanisms together and "hotspots," methods or code regions responsible for a disproportionate share of total energy use, tend to cluster around allocation-heavy loops, poorly bounded collections, and utility methods that get called often but were never optimized. Section V covers the profiling tools used to detect these hotspots, since spotting them by inspection alone stops working once an application has any real size.

7) Technical Challenges. The core technical challenge at the JVM level is that these mechanisms interact rather than operate independently. A refactoring that reduces object allocation might simultaneously push up CPU-bound computation to compensate, and the net energy effect depends on which subsystem dominates for a given workload. That workload dependency, more than any single mechanism, is why the same refactoring technique produces inconsistent results across studies and applications. The separate, community-level problem, the absence of standardized evaluation protocols, is addressed in Section VII.

IV. GREEN REFACTORING TECHNIQUES: A CLASSIFICATION

Across the primary studies included in this review, green refactoring techniques for Java cluster into six categories, shown in Fig. 3. This taxonomy draws on a broader refactoring-taxonomy framing that argues a clear, technique-level taxonomy has to come before any meaningful comparison of sustainability outcomes across studies [22].

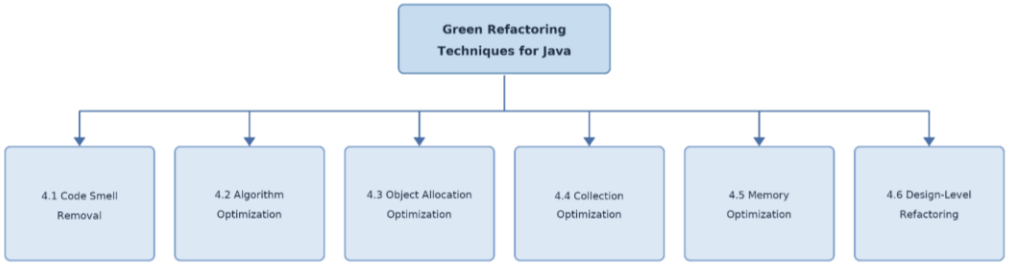

Fig. 3. Six-category taxonomy of green refactoring techniques for Java.

A. Code Smell Removal

The most studied category is removal of classical code smells (long methods, god classes, duplicated code, feature envy), assessed here for energy rather than purely maintainability impact. One of the more rigorous studies in this space refactored a sample of open-source applications and measured energy before and after. Some smell removals cut energy consumption meaningfully; others had negligible or even negative effects, which undercuts any assumption that "cleaner code is automatically greener code" [3]. The same territory has also been approached from the refactoring-catalog side, identifying which specific code smells were most likely to carry an energy penalty and which classical refactorings addressed them [26].

A narrower question inside this category is whether the order in which multiple smells get removed matters. One dissertation tested this directly, running elimination sequences instead of

isolated single-smell refactorings, and found that the order changed the cumulative energy outcome, not just how big the effect was but sometimes which direction it went [6]. That finding deserves more attention than it gets. Most of the literature treats refactorings as independent, additive interventions, but this result suggests treating them as an ordered sequence, where each refactoring changes the conditions the next one operates under, may be closer to how energy behavior actually works.

B. Algorithm Optimization

This category covers swapping an algorithm or data structure for a functionally equivalent but more efficient one, the classic example being an $O(n^2)$ search replaced with an indexed lookup. A cross-language benchmark supplies the underlying evidence that algorithmic choice carries a measurable, sometimes large, energy cost independent of language semantics, and several Java-specific studies in this review lean on its methodology, explicitly or not, as a template for isolating algorithmic effects from other confounds [2].

C. Object Allocation Optimization

Given the heap and garbage collection mechanisms covered in Section III-B, reducing unnecessary object allocation is its own category, separate from general algorithm choice. The Vincent system targets this problem at the JIT level, finding "hot methods" through runtime profiling and applying allocation-reducing optimizations automatically instead of waiting for a developer to spot the pattern by hand [8]. Its results suggest automated detection beats manual code review for finding allocation hotspots, largely because allocation-heavy code doesn't always look wasteful just from reading it.

D. Collection Optimization

Java's collection framework offers multiple implementations (ArrayList vs. LinkedList, HashMap vs. TreeMap) with meaningfully different memory and access-pattern profiles, and picking the wrong one for a given access pattern is a recurring, quietly expensive mistake. The OJXPerf tool takes an object-centric approach to catching this kind of inefficiency, identifying redundant object replicas and collection-level waste that static analysis tends to miss, since the problem often only shows up under a real runtime access pattern [15]. That tool-based detection gives this category an empirical footing that manual code review alone can't match at scale.

E. Memory Optimization

Beyond collections specifically, broader memory optimization covers heap sizing, object pooling, and cutting peak memory footprint. Foundational heap profiling work in this area, though it was never framed around "green" software, underpins much of the tooling energy researchers now use to detect memory-driven waste [12]. That work predates the green software engineering framing by two decades, and that gap is telling: much of the tooling this field depends on was originally built for performance and footprint problems, with energy efficiency retrofitted on afterward.

F. Design-Level Refactoring

The categories above operate at the method or object level. Design-level refactoring targets structural patterns instead. One study of the Visitor design pattern offers the sharpest illustration in this review's sample, comparing Java and C++ implementations of the same pattern and measuring energy differences attributable to the language runtime rather than the pattern's logical structure [9]. The Java implementation carried a consistent energy overhead relative to the C++ version, tied specifically to the extra indirection Java's dispatch mechanism requires, a language-runtime effect rather than a flaw in the pattern itself. That distinction matters, because it shows "design pattern X is expensive" claims need to say which language and runtime they were measured on: the same pattern can behave very differently depending on how the underlying language handles method dispatch and object layout.

G. Comparative Analysis of Techniques

Table II summarizes the six categories side by side, including their typical energy mechanism,

representative studies, and reported consistency of effect across the literature.

TABLE II. COMPARISON OF GREEN REFACTORING TECHNIQUES

Category	Primary Mechanism	Repr. Studies	Reported Consistency
Code smell removal	Reduced method complexity, fewer redundant operations	[3], [6], [26]	Mixed; depends on smell type/sequence
Algorithm optimization	Reduced computational complexity	[2]	Consistent, largest effect sizes
Object allocation optimization	Reduced heap pressure and GC frequency	[8]	Consistent within JIT-targeted contexts
Collection optimization	Reduced object replication and memory access cost	[15]	Consistent, needs tool-based detection
Memory optimization	Reduced peak footprint and collection overhead	[12]	Consistent for footprint, indirect for energy
Design-level refactoring	Reduced dispatch/indirection overhead	[9]	Consistent within pattern, varies by language

V. RUNTIME PROFILING TECHNIQUES

A. Why Profiling Matters for Green Refactoring

Static code analysis can flag candidate refactoring targets, but it can't confirm whether a refactoring actually cut energy use. Only runtime measurement can do that. This section covers the profiling tools used across the primary studies in this review to make that measurement, and how they differ in which layer of the system they observe.

B. Runtime Metrics Monitored

Across the tools discussed below, five metric categories keep coming up: CPU time, energy draw (via hardware counters or OS-level power interfaces), heap allocation, garbage collection frequency and pause time, and method-level invocation counts. Which of these a tool captures determines what kind of refactoring question it can actually answer. A tool that only reports CPU time, for example, can't tell you whether an observed improvement is genuinely energy-related or just a timing artifact.

C. Profiling Tools

VisualVM. A general-purpose JVM monitoring tool bundled with the JDK, VisualVM provides heap, thread, and CPU sampling but no direct energy measurement. It's useful for diagnosing the mechanism behind an energy change, just not for measuring the energy itself.

JFR (Java Flight Recorder). Built into the JVM with low overhead, JFR captures detailed runtime events, including allocation and GC events, at overhead levels safe enough for production. That makes it a common choice for profiling live or production-adjacent workloads rather than isolated benchmark runs.

async-profiler. async-profiler is a low-overhead sampling profiler that captures CPU and allocation profiles through native sampling rather than JVM instrumentation. It gets used often where JFR's event model is too coarse for method-level hotspot detection.

JMC (Java Mission Control). JMC is the analysis and visualization companion to JFR, turning raw JFR recordings into flame graphs and timeline views. It's usually discussed alongside JFR rather than as its own measurement mechanism.

JoularJX. Unlike the four tools above, JoularJX measures energy directly instead of inferring it from CPU or allocation proxies, using a software-based power model to attribute energy consumption down

to the method level inside a running JVM process. The work introducing PowerJoular and JoularJX established this as a multi-platform approach to software power monitoring, and it's the tool cited most often across the empirical studies in this review whenever a paper reports Joule-level, method-attributed figures rather than CPU-time proxies [1]. That difference, direct measurement versus inference, is probably the single most important methodological fact in this section: a study built on CPU-time proxies and one built on JoularJX-style direct measurement are not really answering the same question, even when both report similar-sounding "energy reduction" percentages.

At least one recent approach skips agent-based profiling entirely and takes a different route. Instead of instrumenting the running JVM, "Energy Codesumption" uses existing test suite execution as the vehicle for source-code energy analysis, treating tests that already exist for correctness purposes as a free source of repeatable, comparable energy measurement runs [7]. That's a different trade-off from the five tools above: it gives up fine-grained method attribution in exchange for reusing infrastructure most Java projects already have, which lowers the barrier to adoption compared with standing up a dedicated profiling pipeline.

D. Comparative Analysis of Profiling Tools

Table III consolidates the tool comparison, combining measurement layer, overhead, and metric coverage into a single reference table.

TABLE III. COMPARISON OF RUNTIME PROFILING TOOLS

Tool	Measurement Layer	Direct Energy Meas.	Overhead	Primary Metrics	Best Suited For
VisualVM	JVM (sampling)	No	Low–moderate	CPU, heap, threads	Interactive diagnosis, development use
JFR	JVM (native events)	No	Very low	Allocation, GC, CPU events	Production-safe continuous profiling
async-profiler	Native sampling	No	Low	CPU, allocation flame graphs	Method-level hotspot detection
JMC	Analysis layer over JFR	No	N/A (post-processing)	Timeline, flame graph visualization	Analysis of JFR recordings
JoularJX	Software power model	Yes	Low–moderate	Method-level Joules, CPU energy	Direct, method-attributed energy studies

E. Challenges of Runtime Profiling

Profiling tool choice is not a neutral methodological detail; it affects reported results directly. An empirical study of Java profiler precision and accuracy, pointedly titled "Don't Trust Your Profiler," found meaningful disagreement between different profilers observing the same running program, with some tools systematically over- or under-attributing time and resource usage to specific methods depending on their sampling strategy [14]. That has direct consequences for the green refactoring literature. If two studies use different profilers and report different energy effects for what looks like the same refactoring technique, part of that disagreement may be a profiler artifact rather than a real difference in the refactoring's effect. It's a strong argument for the standardized evaluation protocols discussed later in Section VII: without a shared, validated measurement baseline, cross-study comparison in this field will keep producing results that look contradictory partly because the studies were never actually measuring the same thing.

VI. COMPARATIVE ANALYSIS OF EXISTING STUDIES

A. Comparative Analysis of Techniques

Table IV merges the technique-level comparison with the study-level comparison into a single wide table, since the two views, "which technique" and "which study," largely describe the same set of primary sources from different angles.

TABLE IV. COMPARATIVE ANALYSIS OF GREEN REFACTORING TECHNIQUES AND CORRESPONDING STUDIES

Study	Technique Category	Application Type	Reported Effect	Consistency Across Sample
Verdecchia et al. [3]	Code smell removal	Open-source Java applications	Mixed: some reductions, some negligible/negative	Low; per-application variation
Mehra et al. [19]	Code smell / pattern-based refactoring	Industry codebase (case study)	Consistent energy reduction reported	High within the single studied system
Dhaka [6]	Code smell elimination sequences	Academic benchmark set	Sequence-dependent effect direction	Low; depends on ordering
Pereira et al. [2]	Algorithm/language-level choice	Cross-language benchmark suite	Large, consistent energy differences	High
Liu et al. [8]	Object allocation (JIT-level)	Hot-method-targeted benchmarks	Consistent reduction in targeted methods	High within targeted scope
Li et al. [15]	Collection/object replica detection	Java programs (ICSE evaluation)	Consistent detection of waste; energy inferred	Moderate
Shaham et al. [12]	Memory/heap profiling	Java programs	Footprint reduction confirmed; energy indirect	Moderate
Connolly Bree & Ó Cinnéide [9]	Design-level (Visitor pattern)	Java vs. C++ implementations	Consistent Java-side overhead	High within pattern

The clearest tension in this table sits between two studies. One industry case study reports consistent energy gains from targeted refactoring inside a single production system [19]; a broader academic sample reports mixed results across applications instead [3]. Both studies are methodologically sound, so the disagreement isn't really about whose measurement is more trustworthy. It's more likely a scope difference: a single, well-understood industry system gives refactoring decisions a stable, well-profiled context to act inside, while a diverse sample of open-source applications introduces workload variation that swamps the signal from any one technique. That's arguably the central reason this field disagrees with itself so often. Most studies are, in effect, answering "does this work for this specific system," and generalizing that answer across systems is where the literature keeps running into trouble.

B. Comparative Analysis of Runtime Metrics

Table V compares which metrics different studies actually measured, since not every study measured energy directly.

TABLE V. COMPARISON OF RUNTIME METRICS USED IN PREVIOUS STUDIES

Study	CPU Time	Direct Energy (J)	Heap/Allocation	GC Frequency	Method-Level Attribution
Noureddine [1]	Yes	Yes	No	No	Yes
Verdecchia et al. [3]	Yes	Yes	No	No	No
Pereira et al. [2]	Yes	Yes	No	No	No
Liu et al. [8]	Yes	No (proxy via CPU)	Yes	Partial	Yes
Shaham et al. [12]	No	No	Yes	Partial	No
Maquoi et al. [7]	Yes	Partial	No	No	Partial (test-level)

This inconsistency in what actually gets measured is exactly the fragmentation problem an energy smell taxonomy effort is trying to address. Part of that effort responds to the fact that "energy smell" studies have historically used different metrics for what they each call the same underlying concept, so the taxonomy is, in effect, an attempt to impose shared vocabulary on a fragmented metric landscape [18]. Without that shared vocabulary, a claimed "energy smell" in one study and another study's version of the same smell may not even be measuring the same thing underneath the label.

C. Comparative Analysis of Profiling Tools

The tool-level comparison for this synthesis does not require a new table; it refers back to Table III in Section V-D, which already consolidates measurement layer, overhead, and metric coverage for all five tools discussed in this review.

D. Synthesis of Existing Research

Rather than a fifth comparison table, the cross-cutting pattern across these studies is easier to describe in prose, since it's about scope and architecture rather than a technique-by-technique comparison. Nearly every study reviewed here, code smell papers, allocation papers, design-pattern papers, operates at the level of a single application or a small set of monolithic Java programs. One 2025 study breaks from that pattern more than any other in this review's sample: it looks at energy trade-offs at the microservice architecture level rather than within a single codebase, and finds that architectural tactics meant to improve modularity or resilience often carry their own energy cost, one that offsets, and sometimes outweighs, whatever gets gained from method-level refactoring [5]. It's also the newest and architecturally most different data point in the set, and it points at something the rest of the literature has largely left alone: as Java systems keep getting decomposed into microservices, the unit of analysis for green refactoring research may need to shift from "one method inside one JVM process" to "one call across a network of JVM processes," a much harder measurement problem than anything covered in Sections IV through VI so far.

E. Discussion

Pulling Sections VI-A through VI-D together, the field's central unresolved tension is this: most primary studies work with small samples, report per-application results that don't generalize cleanly, and stay dominated by classical, legacy code smells (long method, god class, feature envy) rather than energy behavior specific to modern Java constructs (streams, lambdas, records, virtual threads). A literature built mostly around smells that predate Java 8 isn't necessarily wrong, but it misses a lot for a language that has kept changing since those smells were first catalogued. Small-N studies, per-application inconsistency, and a legacy-smell-heavy evidence base: that combination is what motivates the research gaps in Section VII.

VII. RESEARCH GAPS AND FUTURE RESEARCH DIRECTIONS

A. Research Gap 1: Absence of Multi-Metric Evaluation

Most studies reviewed here report a single metric, energy, CPU time, or allocation count, rather than tracking several at once. 2026 work on systematically detecting energy regressions makes a strong case for why that matters: a regression that shows up clearly

in one metric can be invisible in another, so single-metric studies risk missing regressions entirely just because they weren't looking at the right signal [20].

B. Research Gap 2: Limited Integration of Profiling and Manual Refactoring

Profiling tools are good at identifying hotspots, but very little tooling connects that identification directly to an automated or semi-automated refactoring suggestion. The test-execution-based approach discussed earlier moves in this direction by tying energy analysis to existing test infrastructure [7]. And the earlier finding that static metrics alone can't reliably predict method-level energy usage explains why this integration matters: without runtime data feeding directly into the refactoring decision, developers are left guessing which static-analysis flags are actually worth acting on [17].

C. Research Gap 3: Lack of Standardized Evaluation Protocols

As Section VI-B already showed, studies measure different things in different ways, which makes cross-study comparison unreliable. A comparative study of green coding adoption across organizations found that practitioners, not just researchers, lack a shared standard for what "green coding practice" even means, so the standardization gap runs on both the academic and industry sides [11]. The GSMM reference model discussed earlier is the most developed attempt in this review's sample to close that gap, but its adoption across the primary studies here remains limited [23].

D. Research Gap 4: Manual Green Refactoring Remains Under-Researched

Automated and tool-detected refactoring (allocation optimization, collection optimization) has a reasonably developed evidence base. Manually applied green refactoring, the kind a developer would do during ordinary code review without specialized tooling, is comparatively under-studied. The sequence-dependent findings discussed earlier are a rare exception that specifically targets manual, developer-driven refactoring decisions rather than tool-automated ones [6], and how little comparable work exists in this category is itself a notable gap.

E. Research Gap 5: Limited Understanding of JVM Runtime Characteristics Under Refactoring

Even where energy effects are measured, the underlying JVM-level explanation for why a given refactoring produced that effect often goes unexplored. The heap profiling foundations and hot-method-targeted optimization work discussed earlier both point toward the mechanisms involved [12], [8], but connecting specific refactoring categories to specific, well-characterized JVM runtime behavior, which GC algorithm, which JIT compilation tier, which escape-analysis outcome, remains open and largely unaddressed across the broader sample.

F. Research Gap 6: Absence of Long-Term Evaluation

Nearly every study in this review measures energy effects immediately after refactoring, in a single evaluation run or a short benchmark session. Work on the ripple effects of refactoring is a rare exception, looking at how a refactoring's consequences propagate through a codebase over time. It raises a question this literature has largely dodged: does a refactoring's energy benefit hold up as the codebase keeps evolving around it, or does it erode as later changes interact with the refactored code in ways the original study never measured [21]?

TABLE VI. RESEARCH GAP ANALYSIS

Gap	Description	Evidence
1. Multi-metric evaluation	Studies rely on single metrics, risking missed regressions	[20]
2. Profiling–refactoring integration	Hotspot detection rarely feeds automated refactoring	[7], [17]
3. Standardized evaluation	No shared protocol across academic and industry practice	[11], [23]
4. Manual refactoring under-studied	Automated/tool-detected techniques dominate the evidence base	[6]
5. JVM runtime characterization	Effects measured without clear mechanistic explanation	[12], [8]
6. Long-term evaluation	Nearly all studies measure only immediate, single-run effects	[21]

Fig. 4 maps each of these six gaps onto the corresponding future direction proposed below.

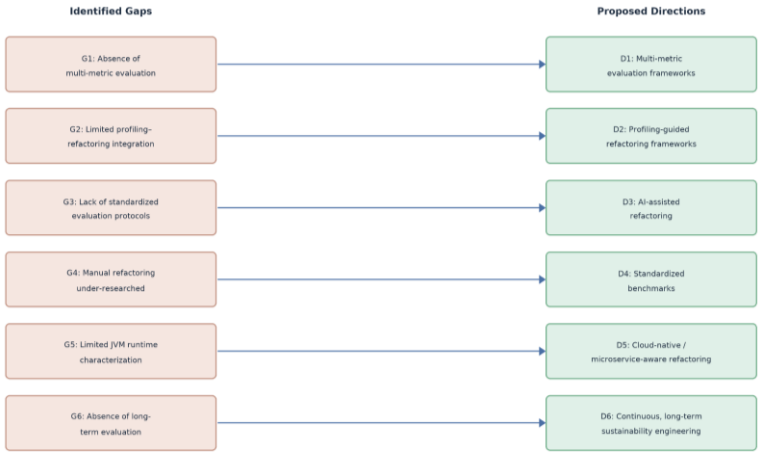

Fig. 4. Mapping of identified research gaps to proposed future directions.

G. Future Research Direction 1: Multi-Metric Evaluation Frameworks

Future work should adopt evaluation frameworks that report energy, CPU time, allocation, and GC behavior together rather than in isolation, following the regression-detection logic proposed earlier, so a technique's effect on one dimension can't silently mask a regression on another [20].

H. Future Research Direction 2: Profiling-Guided Refactoring Frameworks

Tooling that closes the loop between runtime hotspot detection and concrete refactoring suggestions would directly address Gap 2, building on the groundwork already laid by test-execution-based energy analysis and static-plus-dynamic metric combinations appearing in recent literature [7], [17].

I. Future Research Direction 3: AI-Assisted Refactoring

Large language models and AI-assisted development tools are already reshaping how refactoring decisions get made in practice. A mixed-method study of AI's impact across the software development lifecycle on sustainability outcomes suggests this influence extends to energy-relevant decisions, not just functional ones [13]. Whether AI-assisted suggestions systematically favor energy-efficient patterns, or systematically ignore them since energy is rarely part of a model's training signal, is an open and practically urgent question.

J. Future Research Direction 4: Standardized Benchmarks for Green Refactoring

Adopting shared benchmark suites, in the spirit of Renaissance for general JVM workloads but curated specifically around known energy-relevant code patterns, would directly support Gap 3 and make cross-study comparison meaningful instead of approximate [16].

K. Future Research Direction 5: Cloud-Native and Microservice-Aware Green Refactoring

Microservice-level findings discussed earlier already show this direction emerging in the literature [5], and it deserves deliberate expansion given how much production Java now runs as microservices rather than monoliths. Refactoring guidance built only for single-process applications will increasingly miss where the actual energy cost lives.

L. Future Research Direction 6: Continuous, Long-Term Sustainability Engineering

Rather than one-off refactoring evaluations, future studies should track energy behavior continuously as codebases evolve. This would directly address Gap 6, building on the ripple-effect framing discussed earlier to understand whether green refactoring benefits are durable or transient [21].

M. Discussion

These six gaps are not independent of each other. The absence of standardized evaluation (Gap 3) makes multi-metric evaluation (Gap 1) harder to compare across studies even where it is attempted, and the absence of long-term evaluation (Gap 6) means a well-measured, single-run energy

improvement says nothing about whether it survives the codebase's next six months of ordinary development. Closing any one gap in isolation would help. Closing several together would let this field finally produce results that generalize beyond the single application each study happens to have measured.

VIII. CONCLUSION

This review set out to answer a narrower question than most sustainability literature asks: not whether software can be made greener in general, but specifically what is known about refactoring Java code for energy efficiency, and how reliably that knowledge has actually been established. Working from a PRISMA-guided search across five databases and a structured screening protocol, it classified green refactoring into six technique categories, compared five runtime profiling tools (plus one test-execution-based alternative), and synthesized findings across the resulting body of empirical work. The clearest pattern to emerge is that the field agrees code-level decisions matter but disagrees, often for defensible methodological reasons, on how much, under what conditions, and using which measurement approach. Small sample sizes, inconsistent metrics, and a literature still weighted toward classical, pre-Java-8 code smells leave real gaps, particularly around automated profiling-guided refactoring, standardized evaluation protocols, and evaluation that goes beyond a single benchmark run. The six research gaps and six future directions proposed here are meant as a starting agenda, not a finished answer. As Java systems keep shifting toward microservice architectures and AI-assisted development, the questions this review raises, what to measure, how to measure it consistently, and whether today's energy gains survive tomorrow's code changes, are likely to matter more, not less, in the years ahead.

REFERENCES

[1] A. Noureddine, "PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools," in Proc. 18th Int. Conf. Intelligent Environments (IE), Biarritz, France, Jun. 2022, doi: 10.1109/IE54923.2022.9826760.

[2] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy Efficiency across Programming Languages," in Proc. ACM SIGPLAN Int. Conf. Software Language Engineering (SLE), Vancouver, BC, Canada, Oct. 2017, pp. 256–267, doi: 10.1145/3136014.3136031.

[3] R. Verdecchia, R. Aparicio Saez, G. Procaccianti, and P. Lago, "Empirical Evaluation of the Energy Impact of Refactoring Code Smells," in Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME), 2018.

[4] D. Connolly Bree and M. Ó Cinnéide, "How Software Design Affects Energy Performance: A Systematic Literature Review," J. Softw. Evol. Process., vol. 37, no. 1, Art. no. e70014, 2025, doi: 10.1002/smr.70014.

[5] X. Xiao, C. Gao, and J. Bogner, "On the Effectiveness of Microservices Tactics and Patterns to Reduce Energy Consumption: An Experimental Study on Trade-Offs," arXiv:2501.14402, 2025.

[6] G. Dhaka, "An Empirical Investigation into Code Smells Elimination Sequences for Energy Efficient Software," M.Tech. dissertation, Dept. Comput. Sci. Eng., Dr. B. R. Ambedkar National Institute of Technology, Jalandhar, India, Jul. 2016.

[7] J. Maquoi, M. Cauz, B. Vanderose, and X. Devroey, "Energy Codesumption, Leveraging Test Execution for Source Code Energy Consumption Analysis," in Proc. 33rd ACM Int. Conf. Foundations of Software Engineering Companion (FSE Companion), Trondheim, Norway, Jun. 2025, doi: 10.1145/3696630.3728707.

[8] K. Liu, K. Mahmoud, J. Yoo, and Y. D. Liu, "Vincent: Green Hot Methods in the JVM," Sci. Comput. Program., preprint, 2023.

[9] D. Connolly Bree and M. Ó Cinnéide, "Energy Efficiency of the Visitor Pattern: Contrasting Java and C++ Implementations," Empirical Softw. Eng., vol. 28, Art. no. 145, 2023, doi: 10.1007/s10664-023-10387-8.

[10] K. Lano, Z. Rahman, and L. Alwakeel, "Model-Based Software Sustainability Analysis and Improvement," Softw. Syst. Model., 2026, doi: 10.1007/s10270-026-01374-w.

[11] I. Moktan, M. A. Khan, S. Oyedeji, M. Puonti, M. O. Adisa, and J. Porras, "Practical Adoption of Green Coding: A Comparative Study Across Organizations in Finland and Globally," in Business Process Management Workshops, Lecture Notes in Business Information

Processing, vol. 574, 2026, pp. 361–378, doi: 10.1007/978-3-032-14518-5_28.

[12] R. Shaham, E. K. Kolodner, and M. Sagiv, "Heap Profiling for Space-Efficient Java," in Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI), Snowbird, UT, USA, Jun. 2001, pp. 104–113.

[13] F. Iqbal, M. A. Khan, O. Weerakoon, S. Oyedeji, and J. Porras, "The Impacts of Artificial Intelligence throughout the Software Development Life-Cycle on Sustainability: A Mixed-Method Study," in Proc. 10th Int. Workshop on Green and Sustainable Software (GREENS '26), Rio de Janeiro, Brazil, Apr. 2026, doi: 10.1145/3786148.3788622.

[14] H. Burchell, O. Larose, S. Kaleba, and S. Marr, "Don't Trust Your Profiler: An Empirical Study on the Precision and Accuracy of Java Profilers," in Proc. 20th ACM SIGPLAN Int. Conf. Managed Programming Languages and Runtimes (MPLR '23), Cascais, Portugal, Oct. 2023, doi: 10.1145/3617651.3622985.

[15] B. Li, H. Xu, Q. Zhao, P. Su, M. Chabbi, S. Jiao, and X. Liu, "OJXPerf: Featherlight Object Replica Detection for Java Programs," in Proc. IEEE/ACM 44th Int. Conf. Software Engineering (ICSE), Pittsburgh, PA, USA, May 2022, doi: 10.1145/3510003.3510083.

[16] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder, "Renaissance: Benchmarking Suite for Parallel Applications on the JVM," in Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI), Phoenix, AZ, USA, Jun. 2019, doi: 10.1145/3314221.3314637.

[17] M. Imran, V. Stoico, and I. Malavolta, "Static Metrics Are Insufficient: Predicting Java Method Energy Usage with Execution Time," arXiv preprint arXiv:2607.06124, Jul. 2026.

[18] M. Mehditabar, S. Rajput, and T. Sharma, "Watts This Smell: A Comprehensive Taxonomy of Software Energy Smells," 2026. (Preprint).

[19] R. Mehra, P. Pathania, V. S. Sharma, V. Kaulgud, S. Podder, and A. P. Burden, "Assessing the Impact of Refactoring Energy-Inefficient Code Patterns on Software Sustainability: An Industry Case Study," in Proc. IEEE/ACM Int. Conf. Automated Software Engineering (ASE), 2023, doi: 10.1109/ASE56229.2023.00205.

[20] F. Bechet, J. Maquoi, L. Cruz, B. Vanderose, and X. Devroey, "Systematic Detection of Energy Regression and Corresponding Code Patterns in Java Projects," arXiv preprint arXiv:2604.19373, Apr. 2026.

[21] M. Robredo, M. Esposito, F. Palomba, R. Peñaloza, and V. Lenarduzzi, "Analyzing the Ripple Effects of Refactoring," Empirical Software Engineering, vol. 31, Art. no. 134, 2026, doi: 10.1007/s10664-025-10801-3.

[22] A. Almogahed, H. Mahdin, S. Badreddine, M. Omar, A. Alawadhi, S. O. Barraood, M. Othman, S. M. Shaharudin, and S. S. Yaacob, "Towards an Effective Refactoring Taxonomy for Sustainable Software Systems," PLoS One, vol. 21, no. 1, Art. no. e0336296, Jan. 2026, doi: 10.1371/journal.pone.0336296.

[23] A. Guldner, R. Bender, C. Calero, G. S. Fernando, M. Funke, J. Gröger, L. M. Hilty, J. Hörnschemeyer, G.-D. Hoffmann, D. Junger, T. Kennes, S. Kreten, P. Lago, F. Mai, I. Malavolta, J. Murach, K. Obergöker, B. Schmidt, A. Tarara, J. P. De Veaugh-Geiss, S. Weber, M. Westing, V. Wohlgemuth, and S. Naumann, "Development and Evaluation of a Reference Measurement Model for Assessing the Resource and Energy Efficiency of Software Products and Components—Green Software Measurement Model (GSMM)," Future Generation Computer Systems, 2024, doi: 10.1016/j.future.2024.01.033.

[24] A. Pazienza, G. Baselli, D. C. Vinci, and M. V. Trussoni, "A Holistic Approach to Environmentally Sustainable Computing," Innovations in Systems and Software Engineering, vol. 20, pp. 347–371, 2024, doi: 10.1007/s11334-023-00548-9.

[25] 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, vol. 222, Art. no. 112303, 2025, doi: 10.1016/j.jss.2024.112303.

[26] R. Sehgal, D. Mehrotra, R. Nagpal, and R. Sharma, "Green Software: Refactoring Approach," Journal of King Saud University – Computer and Information Sciences, vol. 34, no. 8, pp. 4635–4643, 2022, doi: 10.1016/j.jksuci.2020.10.022.

[27] G. A. García-Mireles, M. Á. Moraga, F. García, and C. Calero, "Sustainability in the Field of Software Engineering: A Tertiary Study," ACM

Transactions on Software Engineering and Methodology, vol. 35, no. 5, Art. no. 141, Apr. 2026, doi: 10.1145/3747178.