DevOpsSustain 2026 · co-located with FSE 2026

Can LLMs Make Software Testing Greener?

An Empirical Study on JUnit Test Energy Reengineering

Xutong Liu · **Andy Zaidman**

University of Twente | Delft University of Technology

The Hidden Cost of Testing

15%

of world energy
consumed by ICT (2020)

2.1–3.9%

of global
greenhouse gas emissions

Frequency

test suites
re-run in CI

But developers barely get feedback on test energy consumption!

Our Question

Can LLMs reengineer Java unit tests to make them more energy efficient?

RQ1

Can an LLM effectively reduce energy consumption of unit tests?

Our Question

Can LLMs reengineer Java unit tests to make them more energy efficient?

RQ1

Can an LLM effectively reduce energy consumption of unit tests?

RQ2

How do LLM-generated modifications impact test effectiveness?

Our Question

Can LLMs reengineer Java unit tests to make them more energy efficient?

RQ1

Can an LLM effectively reduce energy consumption of unit tests?

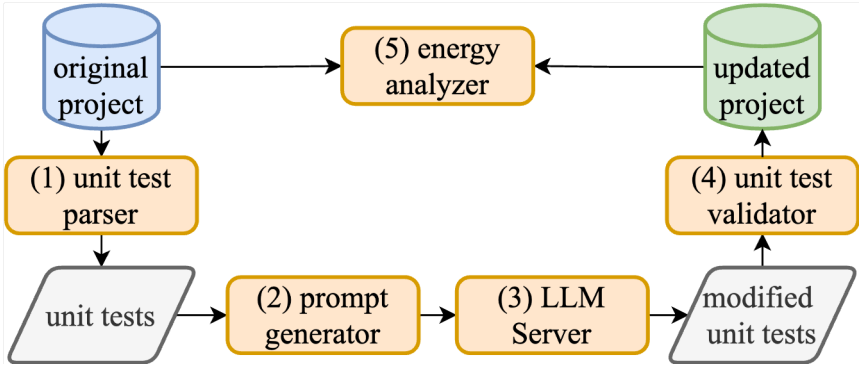
RQ2

How do LLM-generated modifications impact test effectiveness?

RQ3

What types of modifications does the LLM actually make?

A Naïve Methodology



No context; no agent tool; no pre-selection

Dataset Construction

How did we select the 8 subject projects?

1

Input pool

Reproducible open-source Java GitHub projects (Khatami & Zaidman, SPE 2024).

↓

2

Sort

Rank by Test Coverage

↓

3

Filter

Apply Inclusion Criteria

Keep only: (1) Maven build tool (2) JUnit test framework (3) successfully reproducible in our local environment.

→ **First 8 qualifying projects selected (9,705 test methods total)**

Dataset Construction

How did we select the 8 subject projects?

1

Input pool

Reproducible open-source Java GitHub projects (Khatami & Zaidman, SPE 2024).

2

Sort

Rank by Test Coverage

3

Filter

Apply Inclusion Criteria

Keep only: (1) Maven build tool (2) JUnit test framework (3) successfully reproducible in our local environment.

→ **First 8 qualifying projects selected (9,705 test methods total)**

The 8 selected projects

commons-text Text processing	**commons-io** I/O utilities
commons-compress Compression	**cactoos** OOP library
cucumber-reporting Test reports	**lemminx** XML lang. server
rabbitmq-mock RabbitMQ mock	**spring-petclinic** Spring demo

How We Measure Energy

Two categories of tools — each suited to different research questions

Hardware Methods

Best for

- Device-level or system-level studies
- Embedded / IoT energy auditing
- Benchmark comparisons across machines

Not ideal here because

Cannot isolate a single Java method

How We Measure Energy

Two categories of tools — each suited to different research questions

Hardware Methods

Best for

· Device-level or system-level studies

· Embedded / IoT energy auditing

· Benchmark comparisons across machines

Not ideal here because

Cannot isolate a single Java method

Other Software Methods

List:

· Mobile / Android app profiling: PETrA

· Process-level: EnergyBridge

· Python: pyRAPL, pyJoules, CodeCarbon

· Whole-system level: powerstat (no process or method attribution)

Not ideal here because

Cannot isolate the energy of individual test methods within a running JVM.

How We Measure Energy

Two categories of tools — each suited to different research questions

Hardware Methods

Best for

- Device-level or system-level studies
- Embedded / IoT energy auditing
- Benchmark comparisons across machines

Not ideal here because

Cannot isolate a single Java method

Other Software Methods

List:

- Mobile / Android app profiling: PETrA
- Process-level: EnergyBridge
- Python: pyRAPL, pyJoules, CodeCarbon
- Whole-system level: powerstat (no process or method attribution)

Not ideal here because

Cannot isolate the energy of individual test methods within a running JVM.

Our choice

JoularJX

Java method-level energy monitoring

1 ms sampling: finest granularity available

Runs on Linux with Maven

50 runs

RQ1: Energy Reduction Results

Total tests	9,705

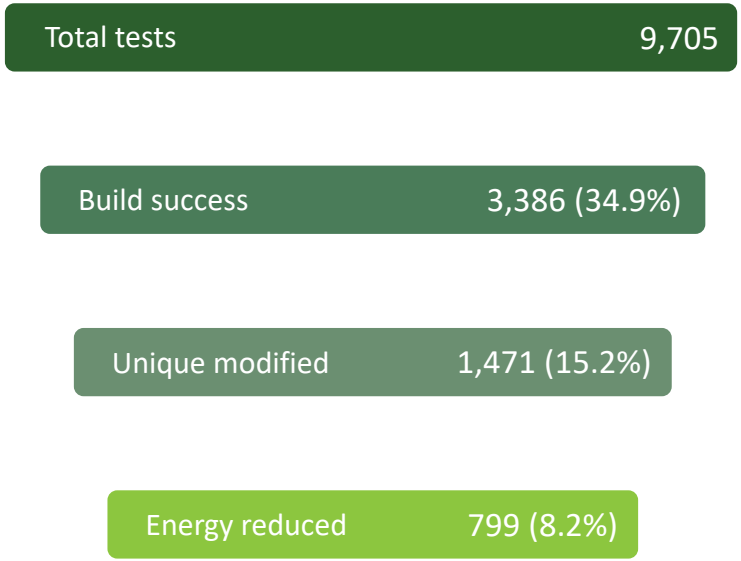
RQ1: Energy Reduction Results

Total tests	9,705
Build success	3,386 (34.9%)

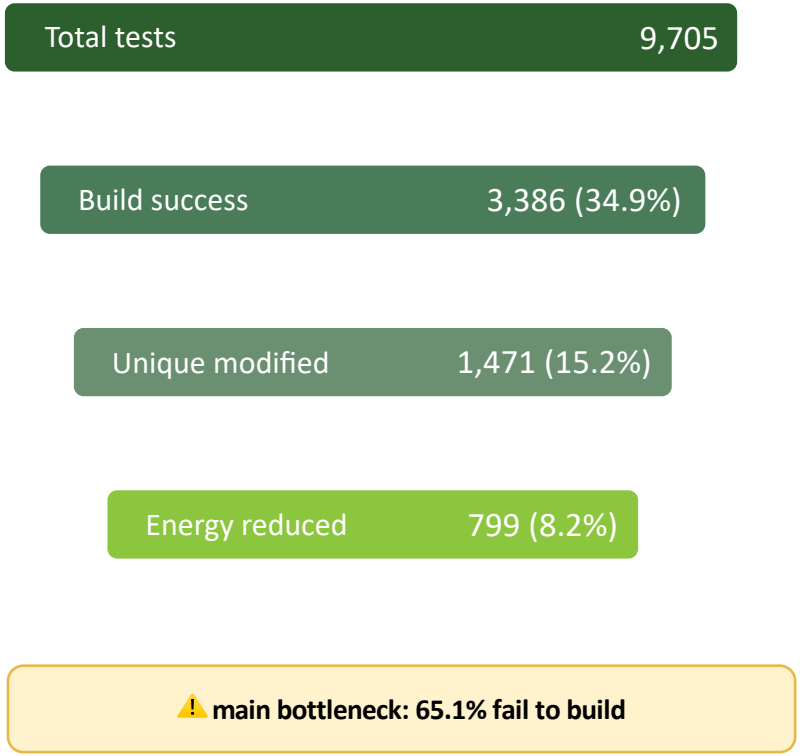
RQ1: Energy Reduction Results

Total tests	9,705
Build success	3,386 (34.9%)
Unique modified	1,471 (15.2%)

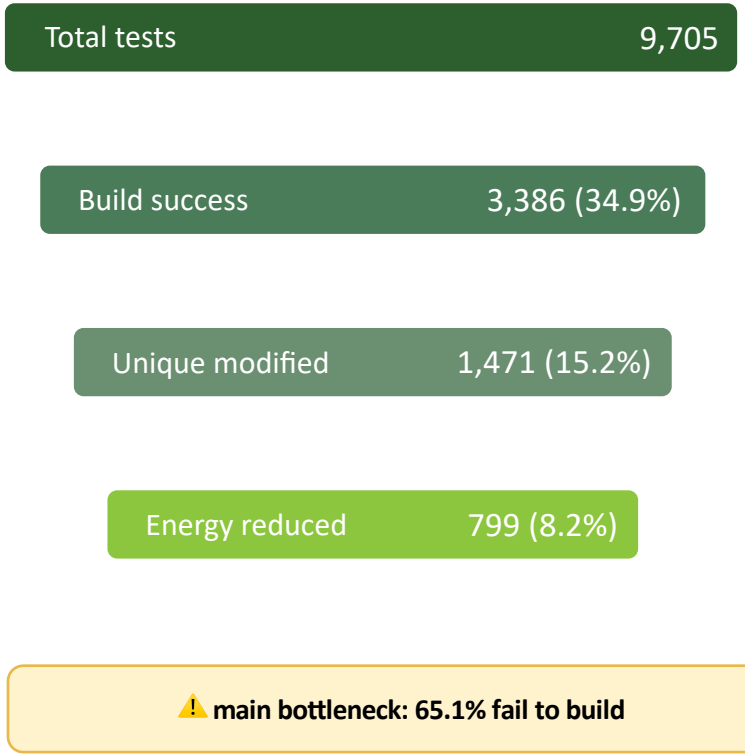
RQ1: Energy Reduction Results



RQ1: Energy Reduction Results



RQ1: Energy Reduction Results



8.2%

tests show improved energy after LLM reengineering

Project-level outcomes

✓ 2 projects: significant energy decrease
✗ 2 projects: significant energy increase
~ 4 projects: no significant change

How much is 1 joule?

≈ Lift a 100 g glass of water about 1 meter upward

Energy = m × g × h
≈ 0.10 kg × 9.8 m/s² × 1 m
≈ 0.98 J ≈ 1 J

How Much Energy Did We Actually Save?

Test-suite level energy (Joules) · original vs updated · 50 runs each · JoularJX

Project	Orig. (J)	Upd. (J)	Diff	p-val	Orig-all (J)	Upd-all (J)	Diff-all
cactoos	128.8	128.9	+0.1%	0.870	398.8	418.0	+4.8%
c-compress	149.2	147.9	−0.9%	0.320	1369.2	1291.5	−5.7%
commons-io	418.5	468.7	+12.0%	0.000★	1945.5	1985.6	+2.1%
c-text	359.3	368.6	+2.6%	0.002★	1179.3	1199.4	+1.7%
cu-reporting	79.7	82.0	+2.9%	0.026★	217.0	239.9	+10.6%
lemminx	48.1	45.5	−5.5%	0.130	1225.8	1193.0	−2.7%
rabbitmq-mock	105.9	105.2	−0.6%	0.310	294.7	293.3	−0.5%
spring-petclinic	84.2	84.9	+0.8%	0.120	759.4	759.3	0.0%

Energy decrease Energy increase (★ = p<0.05)

Best case: c-compress saves 77.7 J per full test run (−5.7%★) · lemminx test scope −2.6 J (−5.5%)
3 projects show significant increases; net benefit is project-dependent

BEST CASE · COMMONS-COMPRESS

The Best Case, Scaled to a Year

commons-compress's −5.7% test-suite saving

~600

CI runs / year

apache/commons-compress, Java CI

77.7 J

saved per run

1,369.2 → 1,291.5 J (−5.7%★)

46.6 kJ

saved / year

≈ 12.95 Wh

1h 26m

of LED bulb light

Assumptions: ~600 CI runs/year estimated from GitHub Actions run history (apache/commons-compress, "Java CI" workflow); saving from JoularJX test-suite measurement (50 runs each, original vs. updated); LED bulb draw of 9W (60W-incandescent-equivalent).

RQ2: Test Effectiveness After Reengineering

Code Coverage

≤ −2%

Instruction coverage drops at most 2%
for most projects. 3 out of 8 projects: unchanged.

RQ2: Test Effectiveness After Reengineering

Code Coverage

≤ −2%

Instruction coverage drops at most 2%
for most projects. 3 out of 8 projects: unchanged.

Mutation Score

≤ −1.16%

Mutation score decreases slightly (−0.21% to −1.16%)
in 6 projects. 2 projects remain unchanged.

RQ2: Test Effectiveness After Reengineering

Code Coverage

≤ −2%

Instruction coverage drops at most 2%
for most projects. 3 out of 8 projects: unchanged.

Mutation Score

≤ −1.16%

Mutation score decreases slightly (−0.21% to −1.16%)
in 6 projects. 2 projects remain unchanged.

✅ LLM modifications have a limited negative impact on fault-detection ability.
⚠️ But this is partly because the modifications themselves are limited in scope.

RQ3: What Does the LLM Actually Do?

Manual analysis of 28 test methods with statistically significant energy improvement ($p < 0.05$, non-negligible effect size)

A

Invalid UUT Invocation (n=4)

Skips or bypasses the Unit Under Test entirely energy saved, but test becomes meaningless

B

Assertion Reduction (n=23)

Removes intermediate/edge-case assertions. Fewer checks, less compute, but weaker tests

C

Syntax-Level Refactoring (n=6)

Simplifies syntax (sugar, variable decls). Preserves logic but often doesn't save a large amount of energy

D

Iteration Optimization (n=2)

Reduces loop counts or replaces expensive inner operations. Large savings but may miss rare bugs

E

Concurrency Optimization (n=2)

Parallelizes thread logic. Reduces time but less reproducible

RQ3: What Does the LLM Actually Do?

Manual analysis of 28 test methods with statistically significant energy improvement ($p < 0.05$, non-negligible effect size)

A

Invalid UUT Invocation (n=4)

Skips or bypasses the Unit Under Test entirely energy saved, but test becomes meaningless

B

Assertion Reduction (n=23)

Removes intermediate/edge-case assertions. Fewer checks, less compute, but weaker tests

C

Syntax-Level Refactoring (n=6)

Simplifies syntax (sugar, variable decls). Preserves logic but often doesn't save a large amount of energy

D

Iteration Optimization (n=2)

Reduces loop counts or replaces expensive inner operations. Large savings but may miss rare bugs

E

Concurrency Optimization (n=2)

Parallelizes thread logic. Reduces time but less reproducible

Mostly workaround strategies!

B — Assertion Reduction

Most Frequent Category ·

B1 — Removing Intermediate Assertions

LLM drops a redundant check when later assertions already imply it, reduces work but weakens the test contract

What happened?

The original test has 3 assertions verifying that applyPatchForFeatures() swaps total failed values:

(1) total > failed [intermediate check]
(2) current failed = old total
(3) current total = old failed

The LLM removes assertion (1), reasoning that (2) and (3) already imply it.

⚠ **Energy saved, but...**

Intermediate assertions document intent explicitly. Removing them may pass today but hide regressions if assertion (2) or (3) changes.

net.masterthought.cucumber.TrendsTest

```
1  1  @Test
2  2  void
      applyPatchForFeatures_OnFailedGreaterThanTotal_ChangesTotalFeatureAndFailed()
3  3  throws Exception {
4  4
5  5      // given
6  6      final int totalFeatures = 1000;
7  7      final int failedFeatures = totalFeatures + 1;
8  8      Trends trends = new Trends();
9  9      Reportable result = new ReportableBuilder(0, failedFeatures,
10 10     totalFeatures, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3206126182398L);
11 11     trends.addBuild(""buildNumber"", result);
12 12
13 13     // when
14 14     Whitebox.invokeMethod(trends, ""applyPatchForFeatures"");
15 15
16 16     // then
17
18       assertThat(trends.getTotalFeatures()
      [0]).isGreaterThan(trends.getFailedFeatures()[0]);
19       // check if the values were reversed
```

BEST CASE · COMMONS-COMPRESS

The Best Case, Scaled to a Year

commons-compress's −5.7% test-suite saving

~600

CI runs / year

apache/commons-compress, Java CI

77.7 J

saved per run

1,369.2 → 1,291.5 J (−5.7%★)

46.6 kJ

saved / year

≈ 12.95 Wh

1h 26m

of LED bulb light

Assumptions: ~600 CI runs/year estimated from GitHub Actions run history (apache/commons-compress, "Java CI" workflow); saving from JoularJX test-suite measurement (50 runs each, original vs. updated); LED bulb draw of 9W (60W-incandescent-equivalent).

Why Does the LLM Struggle?

Two compounding reasons — both need to be fixed

① Training Data Scarcity

0 / 21,213 commits mention energy reduction in our subject projects

233 Qs energy-tagged on StackOverflow in 17 years, 47.6% score ≤ 0

239 items Across 3.3M+ GitHub commits, 94.6% buried in code comments

Why Does the LLM Struggle?

Two compounding reasons — both need to be fixed

① Training Data Scarcity

0 / 21,213 commits mention energy reduction in our subject projects

233 Qs energy-tagged on StackOverflow in 17 years, 47.6% score ≤ 0

239 items Across 3.3M+ GitHub commits, 94.6% buried in code comments

② Our Approach is Naive

No context

Prompt contains only the test method, no production code, no dependencies, no project knowledge

No validation loop

LLM generates code once; 65% fails to build, no iterative repair or feedback from the compiler

No strategy

Prompt says 'save energy', but gives no hint of how (e.g. mocking, caching). LLM guesses: shortens code

Why Does the LLM Struggle?

Two compounding reasons — both need to be fixed

① Training Data Scarcity

0 / 21,213 commits mention energy reduction in our subject projects

233 Qs energy-tagged on StackOverflow in 17 years, 47.6% score ≤ 0

239 items Across 3.3M+ GitHub commits, 94.6% buried in code comments

② Our Approach is Naive

No context

Prompt contains only the test method, no production code, no dependencies, no project knowledge

No validation loop

LLM generates code once; 65% fails to build, no iterative repair or feedback from the compiler

No strategy

Prompt says 'save energy', but gives no hint of how (e.g. mocking, caching). LLM guesses: shortens code

Fix both: Agentic pipeline + full project context + explicit strategy injection = our next work (submitted)

What's Next · From Slow to Green

From Naive to Strategic

How we addressed each limitation in the follow-up work (submitted)

What's Next · From Slow to Green

From Naive to Strategic

How we addressed each limitation in the follow-up work (submitted)

No context
(test only)

Full project context
+ production code

From Naive to Strategic

How we addressed each limitation in the follow-up work (submitted)

No context
(test only)

No strategy
(just 'save energy')

Full project context
+ production code

Mock-based strategy
(mined from 127 real fixes)

From Naive to Strategic

How we addressed each limitation in the follow-up work (submitted)

No context (test only)	No strategy (just 'save energy')	No validation (65% build fail)
Full project context + production code	Mock-based strategy (mined from 127 real fixes)	Agentic repair loop

What's Next · From Slow to Green

From Naive to Strategic

How we addressed each limitation in the follow-up work (submitted)

No context (test only)	No strategy (just 'save energy')	No validation (65% build fail)
↓	↓	↓
Full project context + production code	Mock-based strategy (mined from 127 real fixes)	Agentic repair loop

Result: statistically significant energy & time reduction across 20 projects · Spearman $r_s = 0.929$ (time vs. energy)

Key Takeaways

Naive prompting ≠ energy optimization

Only 8.2% success. Without domain knowledge, the LLM shortens code rather than truly reducing energy.

Key Takeaways

Naive prompting ≠ energy optimization

Only 8.2% success. Without domain knowledge, the LLM shortens code rather than truly reducing energy.

Training data scarcity is the root cause

Energy-efficiency discussion is absent from SO, GitHub, and project commits. LLMs can't learn what isn't there.

Key Takeaways

Naive prompting ≠ energy optimization

Only 8.2% success. Without domain knowledge, the LLM shortens code rather than truly reducing energy.

Effectiveness impact is low, but for the wrong reasons

Coverage and mutation scores barely changed, largely because the modifications themselves were superficial.

Training data scarcity is the root cause

Energy-efficiency discussion is absent from SO, GitHub, and project commits. LLMs can't learn what isn't there.

Key Takeaways

Naive prompting ≠ energy optimization

Only 8.2% success. Without domain knowledge, the LLM shortens code rather than truly reducing energy.

Effectiveness impact is low, but for the wrong reasons

Coverage and mutation scores barely changed, largely because the modifications themselves were superficial.

Training data scarcity is the root cause

Energy-efficiency discussion is absent from SO, GitHub, and project commits. LLMs can't learn what isn't there.

Strategy matters

The LLM needs to know **how** to reduce energy, not just be asked to. Domain-specific strategy injection is the path forward.

Who Are We?

Xutong Liu

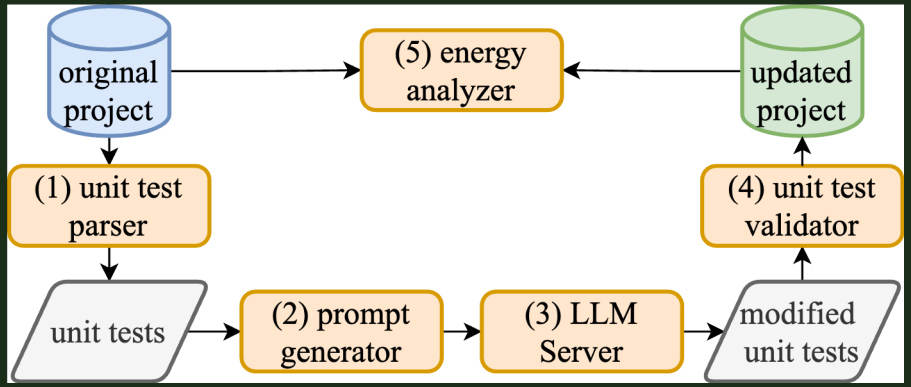
Postdoc | University of Twente

LLM Reliability
LLMs for Software Engineering
Software Testing
Sustainable Software Engineering

Andy Zaidman

Full professor | Delft University of Technology

Software evolution
Software testing
Software reengineering
Sustainable engineering



Thank You

Questions welcome!

Xutong Liu xutong.liu@utwente.nl

https://zenodo.org/records/17279250

Replication package: doi:10.5281/zenodo.17279250

8.2% success · Build failures dominate · LLM does not follow generic prompting · Training data is the bottleneck