FSE Companion '26 · Montréal

The Energy Impact of Batch Testing in CI

An Empirical Study of Static and Dynamic Batching Strategies

Máté Oszkó
BSc Student
TU Delft

Xutong Liu 🎤
Postdoc Researcher
University of Twente

Andy Zaidman
Full Professor
TU Delft

Carolin Brandt
Asst. Professor
TU Delft

The Team

Máté Oszkó

BSc Student · TU Delft

Andy Zaidman

Full Professor · TU Delft

Xutong Liu

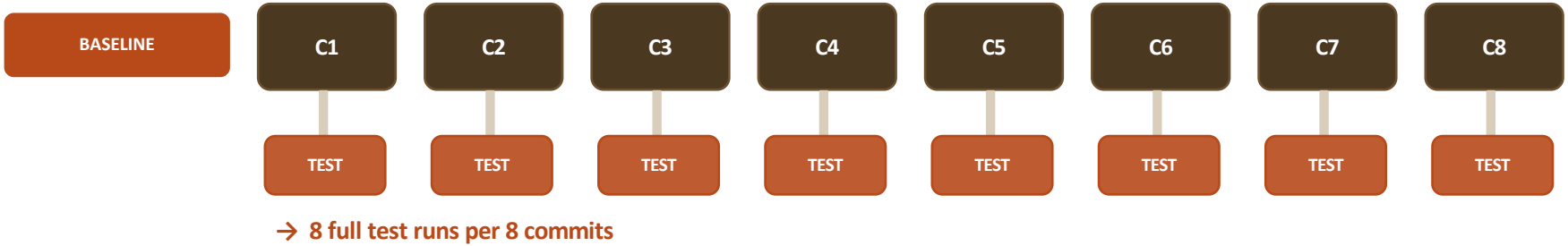
Postdoc University of Twente

Carolin Brandt

Assistant Professor · TU Delft

BACKGROUND

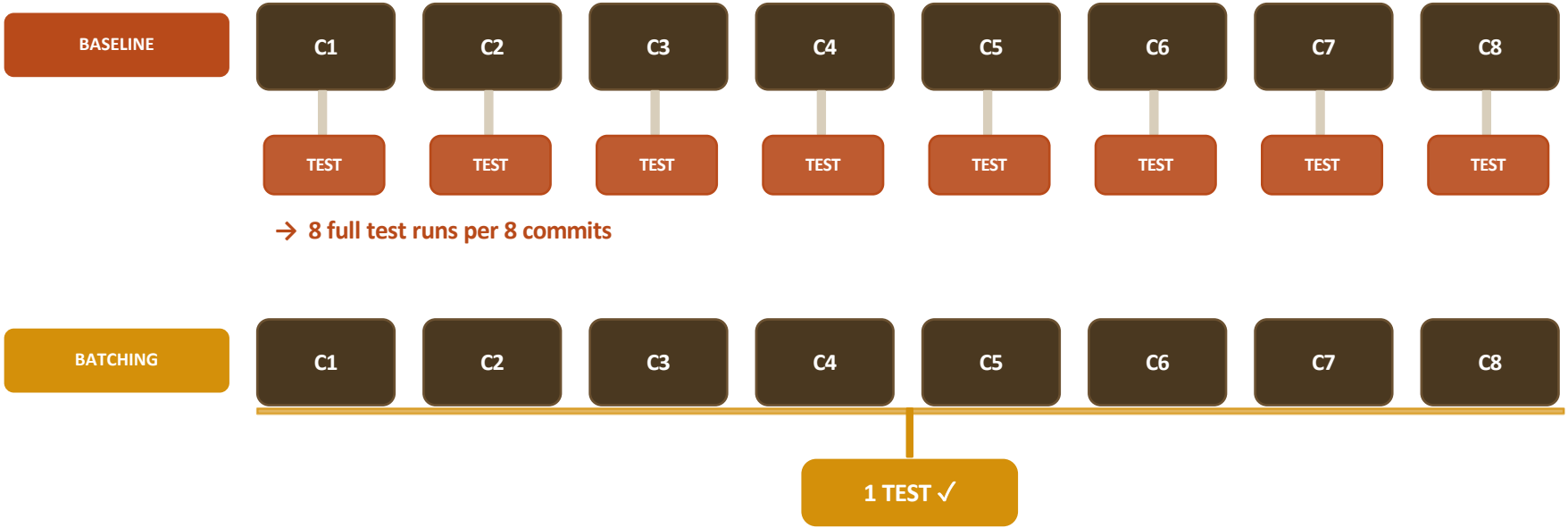
What is Batch Testing?



→ 8 full test runs per 8 commits

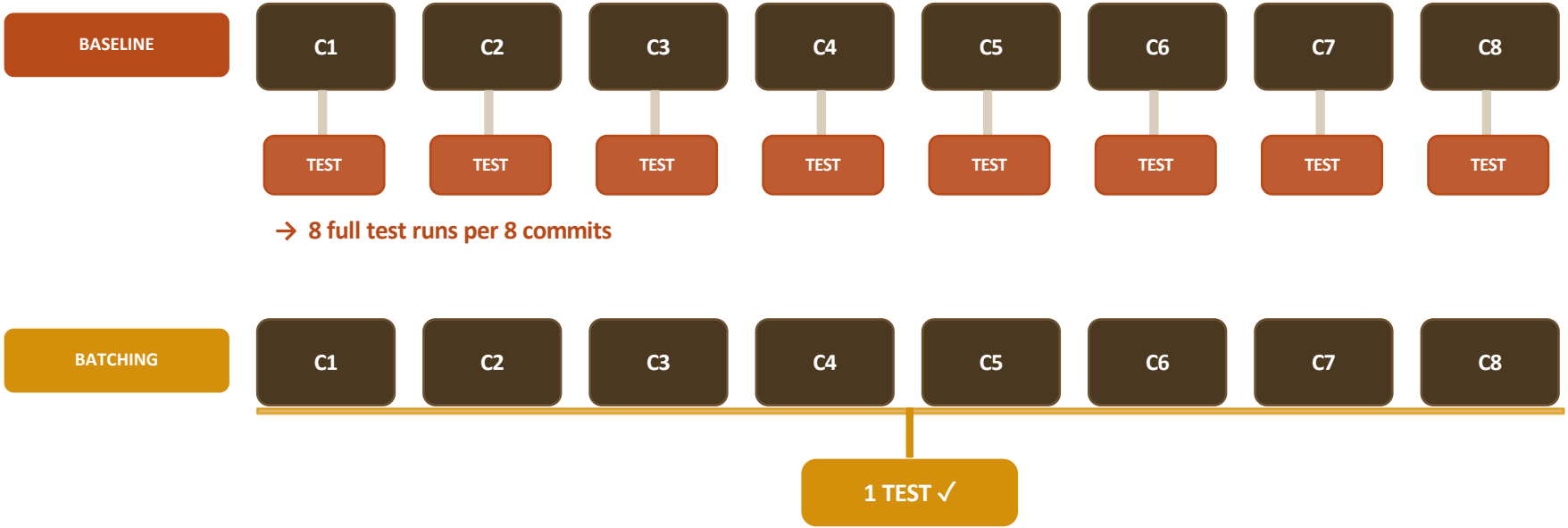
BACKGROUND

What is Batch Testing?



BACKGROUND

What is Batch Testing?



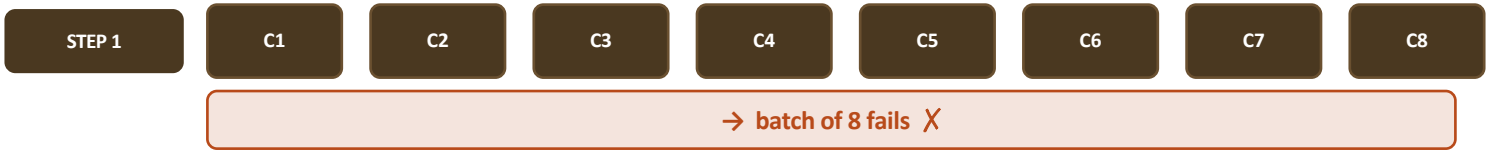
Two strategies studied:

- BatchStop4 (static)
- linear-4 lwd (dynamic)

BACKGROUND

BatchStop4: Static Bisection

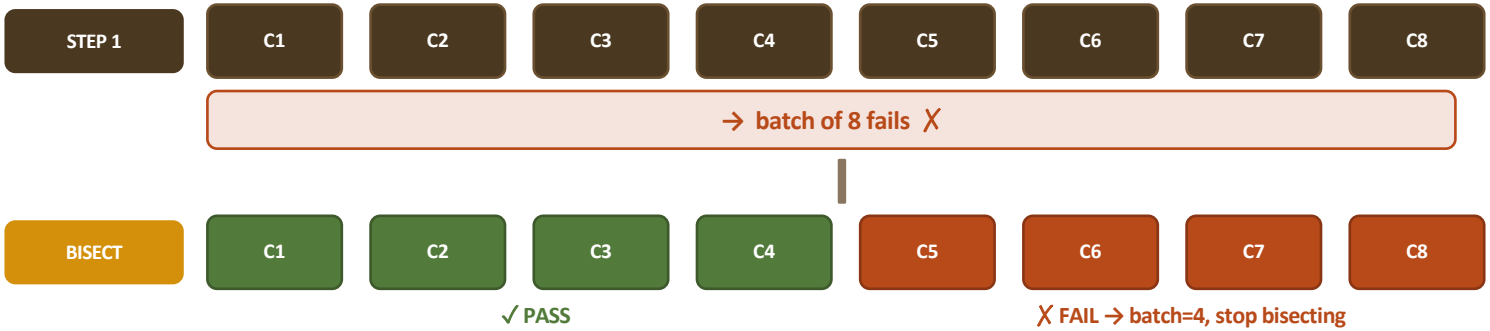
Fixed batch size of 8. On failure, bisect in half repeatedly, until the batch shrinks to 4, then test commits one by one.



BACKGROUND

BatchStop4: Static Bisection

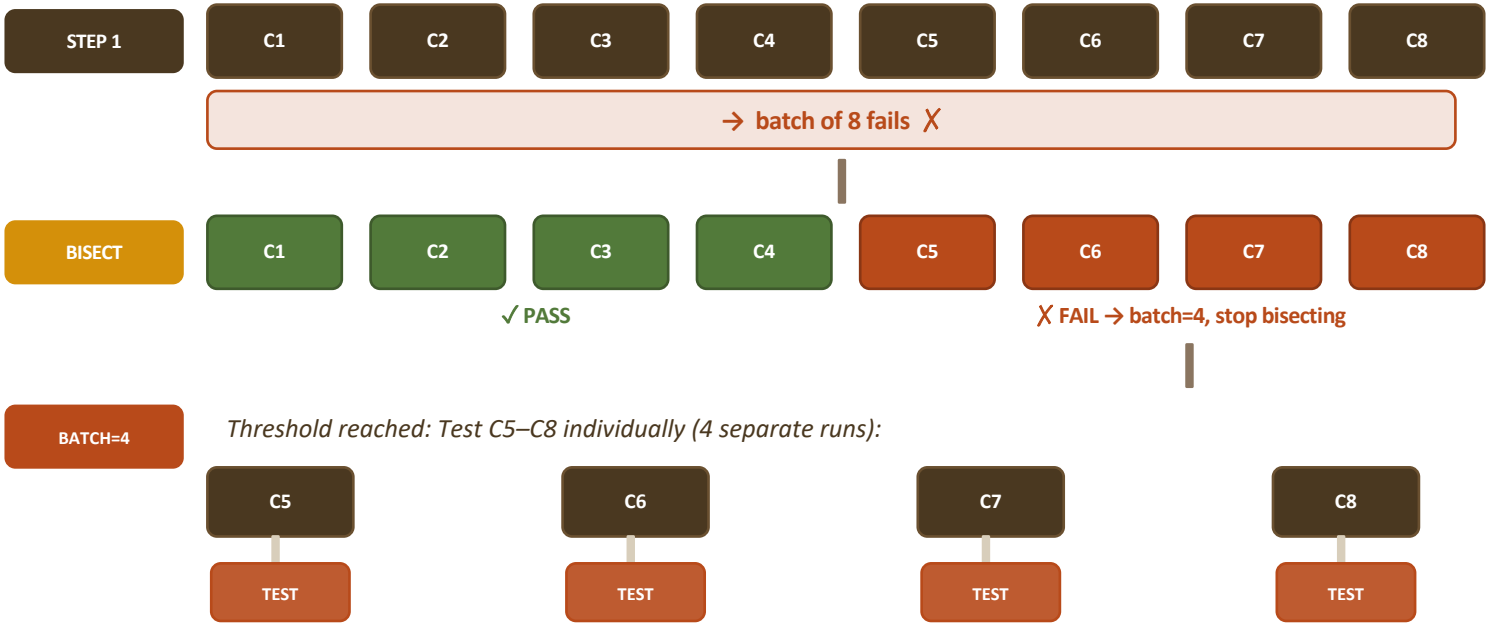
Fixed batch size of 8. On failure, bisect in half repeatedly, until the batch shrinks to 4, then test commits one by one.



BACKGROUND

BatchStop4: Static Bisection

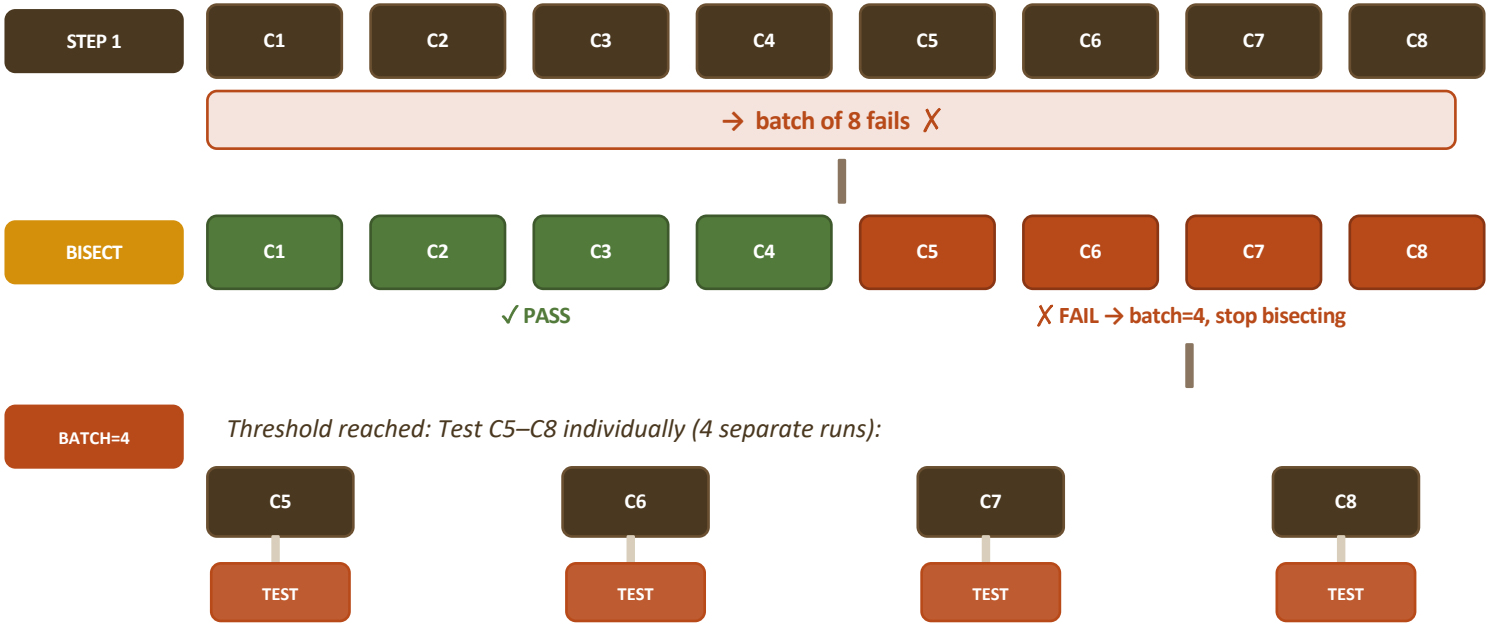
Fixed batch size of 8. On failure, bisect in half repeatedly, until the batch shrinks to 4, then test commits one by one.



BACKGROUND

BatchStop4: Static Bisection

Fixed batch size of 8. On failure, bisect in half repeatedly, until the batch shrinks to 4, then test commits one by one.



Total for this batch: 1 + 2 + 4 = 7 runs (vs. 8 baseline runs), even when a failing commit is present

BACKGROUND

linear-4 lwd: Dynamic Batch Sizing

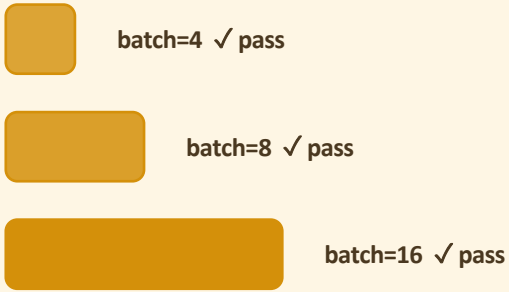
Batch size adapts to the observed failure rate. Stable codebase → batches grow. Frequent failures → batches shrink. On failure, falls back to BatchStop4-style bisection.

BACKGROUND

linear-4 lwd: Dynamic Batch Sizing

Batch size adapts to the observed failure rate. Stable codebase → batches grow. Frequent failures → batches shrink. On failure, falls back to BatchStop4-style bisection.

Stable codebase → batch grows



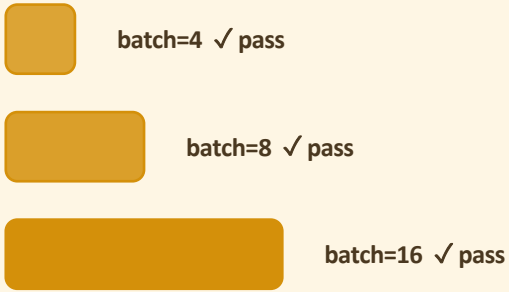
Each pass roughly doubles the batch size, maximising execution savings.

BACKGROUND

linear-4 lwd: Dynamic Batch Sizing

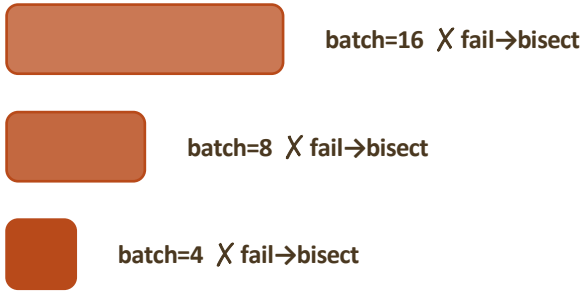
Batch size adapts to the observed failure rate. Stable codebase → batches grow. Frequent failures → batches shrink. On failure, falls back to BatchStop4-style bisection.

Stable codebase → batch grows



Each pass roughly doubles the batch size, maximising execution savings.

Frequent failures → batch shrinks



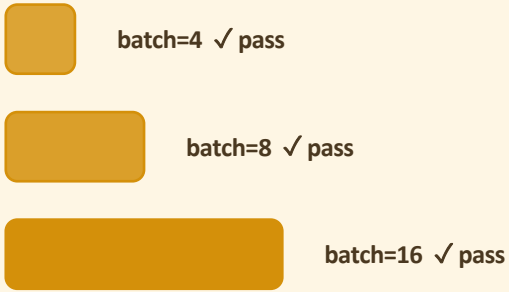
Each failure halves the batch size and triggers bisection, protecting feedback speed.

BACKGROUND

linear-4 lwd: Dynamic Batch Sizing

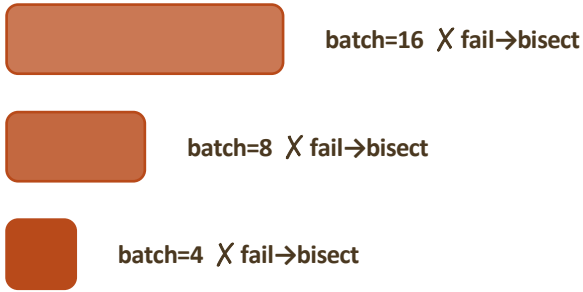
Batch size adapts to the observed failure rate. Stable codebase → batches grow. Frequent failures → batches shrink. On failure, falls back to BatchStop4-style bisection.

Stable codebase → batch grows



Each pass roughly doubles the batch size, maximising execution savings.

Frequent failures → batch shrinks



Each failure halves the batch size and triggers bisection, protecting feedback speed.

Key difference from BatchStop4: batch size is never fixed; it tracks recent build history, balancing feedback speed against execution savings.

RESEARCH GAP

How Prior Work Simulated Batching

How it worked

1. **Take a historical commit log**

   Pull a project's real commit history from GitHub: Which commits passed, which failed, and when.

2. **Replay it on paper**

   Group the commits into hypothetical batches according to the algorithm's rules. No code is built or tested.

3. **Count, don't measure**

   Count how many test executions the batching strategy would have needed, versus the original run-all baseline.

4. **Report the reduction**

   Express the result as a percentage drop in execution count, e.g. '48% fewer test runs.'

No build ever runs. No CPU is ever measured.

Why simulation made sense

SCALE	Analyse thousands of commits across many projects in minutes.
COST	No dedicated hardware.
REPRODUCIBILITY	A static commit log gives the exact same result every time.

The tradeoff: realistic about scale, blind to energy.

RESEARCH GAP

How Prior Work Simulated Batching

How it worked

1. **Take a historical commit log**

   Pull a project's real commit history from GitHub: Which commits passed, which failed, and when.

2. **Replay it on paper**

   Group the commits into hypothetical batches according to the algorithm's rules. No code is built or tested.

3. **Count, don't measure**

   Count how many test executions the batching strategy would have needed, versus the original run-all baseline.

4. **Report the reduction**

   Express the result as a percentage drop in execution count, e.g. '48% fewer test runs.'

No build ever runs. No CPU is ever measured.

Why simulation made sense

SCALE	Analyse thousands of commits across many projects in minutes.
COST	No dedicated hardware.
REPRODUCIBILITY	A static commit log gives the exact same result every time.

The tradeoff: realistic about scale, blind to energy.

Simulation answers "how many fewer runs?" It cannot answer "how many fewer watt-hours?".

OUR APPROACH

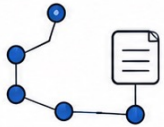
What We Did Instead

OUR EXECUTION PIPELINE

1 Select Repository

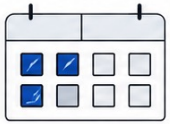
Choose open-source Java projects from GitHub.

2 Collect Full Commit History



Download the complete commit history in chronological order.

3 Apply Batch Strategy (Per Commit)



For each new commit, the batch algorithm decides whether to trigger a build now or wait for more commits.

Strategies: BatchStop4 (static) and linear-4 lwd (dynamic).

4 Execute Real CI Build

When a build is triggered, we checkout the selected commits and run the real CI pipeline and tests (Maven/Gradle) on our CI environment.

5 Measure Energy

EnergiBridge measures CPU package energy consumption during the entire build (Wh).

6 Log Build Outcome

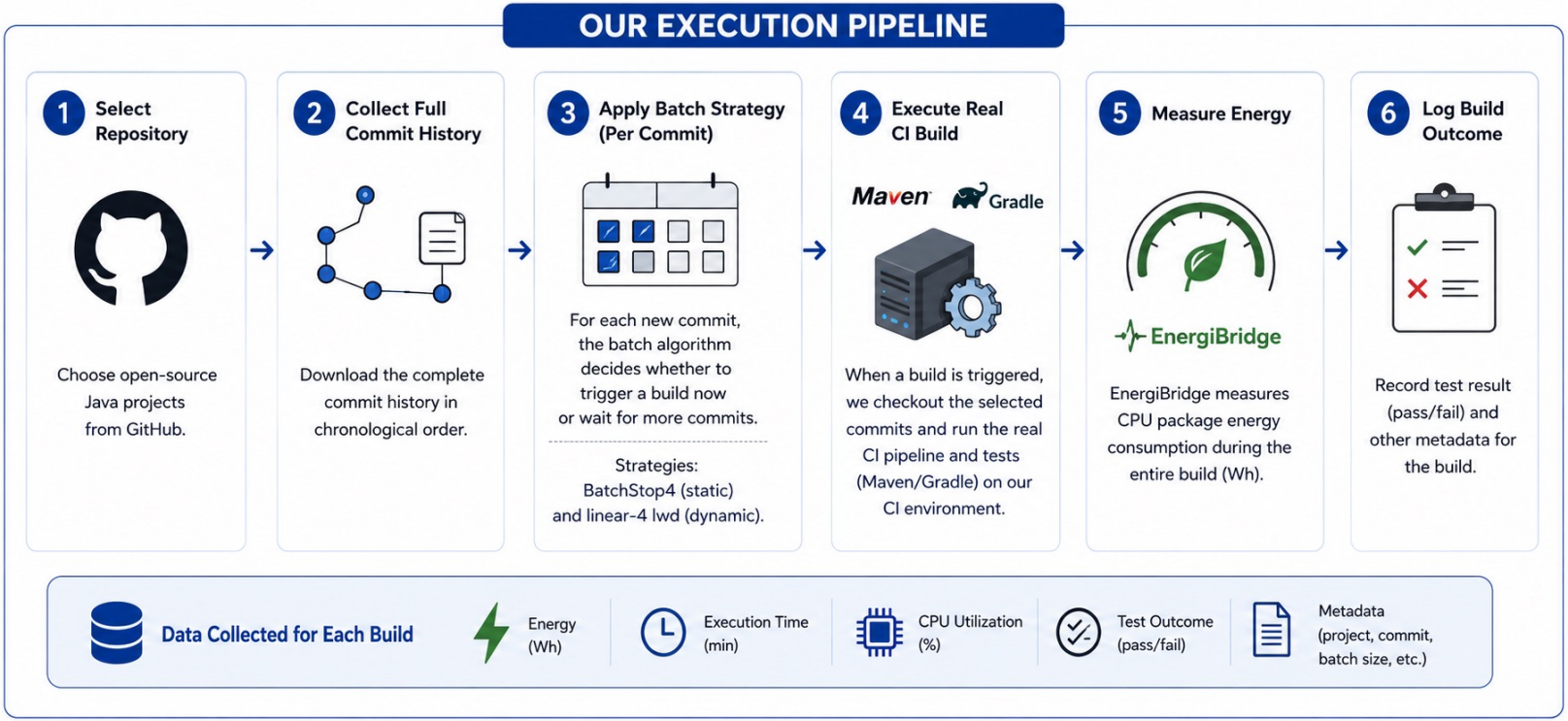
Record test result (pass/fail) and other metadata for the build.

Data Collected for Each Build

- Energy (Wh)
- Execution Time (min)
- CPU Utilization (%)
- Test Outcome (pass/fail)
- Metadata (project, commit, batch size, etc.)

OUR APPROACH

What We Did Instead



Same commits, same algorithms, same projects as prior work, but executed for real, on real hardware, with energy measured directly.

OUR APPROACH

What this buys us

1	**Real energy:**	Joules measured from actual CPU package power draw — not inferred from execution counts.
2	**Real hardware behaviour:**	Captures idle time, caching, and utilisation patterns that a spreadsheet count can never see.
3	**Real failures:**	Bisection actually happens — the cost of finding a culprit commit is paid and measured, not assumed.
4	**A direct answer:**	Energy savings (Wh) and time savings, measured side by side on the same hardware, same builds.

STUDY DESIGN

Research Questions

Two questions guide the empirical study: The magnitude of energy savings, and what drives it.

RQ1

Energy Savings

How much energy do batching strategies save compared to the run-all baseline?

RQ2

Drivers of Savings

What project factors (CPU utilisation, failure rate) explain the differences in savings?

METHODOLOGY

Study Design

1

8 Java Projects

Apache Commons libs
+ Apache Lucene
(Maven & Gradle)

2

160 Commits

Replayed in order
3 repeats each
Flaky excluded

3

EnergiBridge

CPU package energy
per build (Joules)
100 ms sampling

4

Bare Metal

AMD Ryzen 7 9800X3D
48 GB DDR5
Kubuntu 22.04

METHODOLOGY

Study Design

1

8 Java Projects

Apache Commons libs
+ Apache Lucene
(Maven & Gradle)

2

160 Commits

Replayed in order
3 repeats each
Flaky excluded

3

EnergiBridge

CPU package energy
per build (Joules)
100 ms sampling

4

Bare Metal

AMD Ryzen 7 9800X3D
48 GB DDR5
Kubuntu 22.04

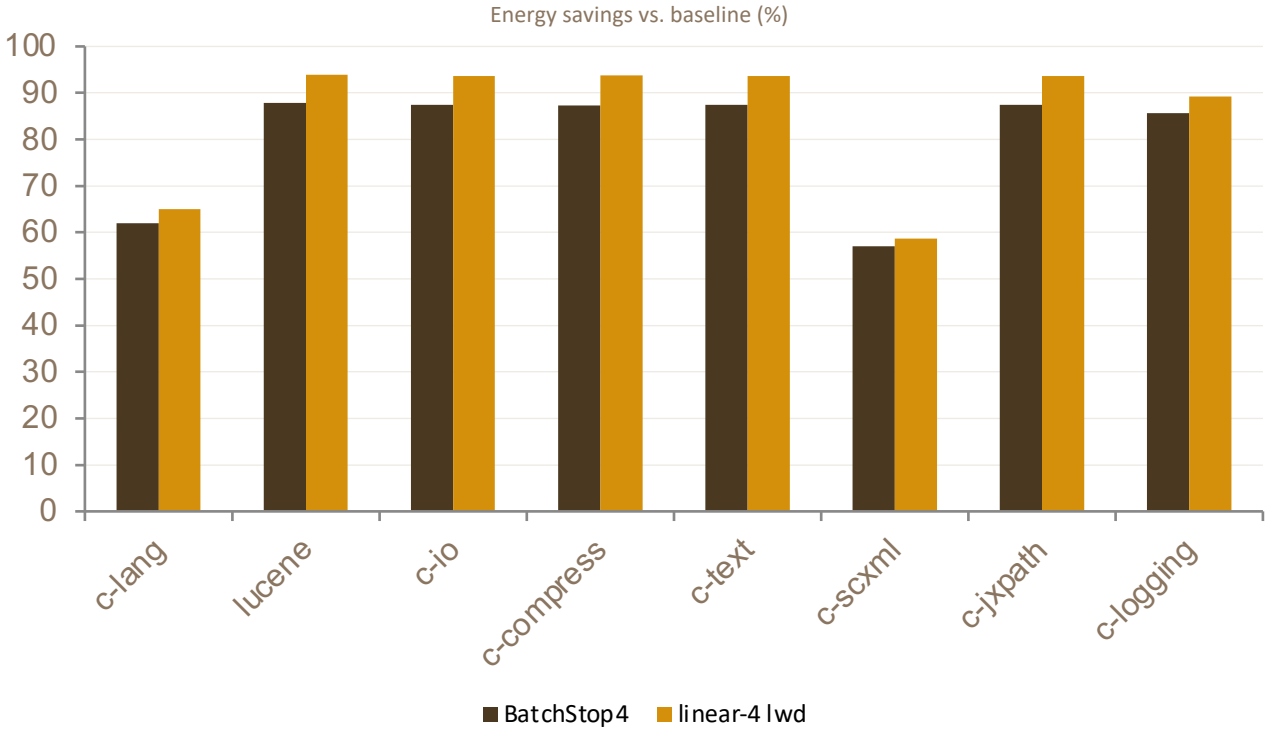
Three configurations compared per project:

Baseline
Run-all every commit

BatchStop4
Static (batch size = 8)

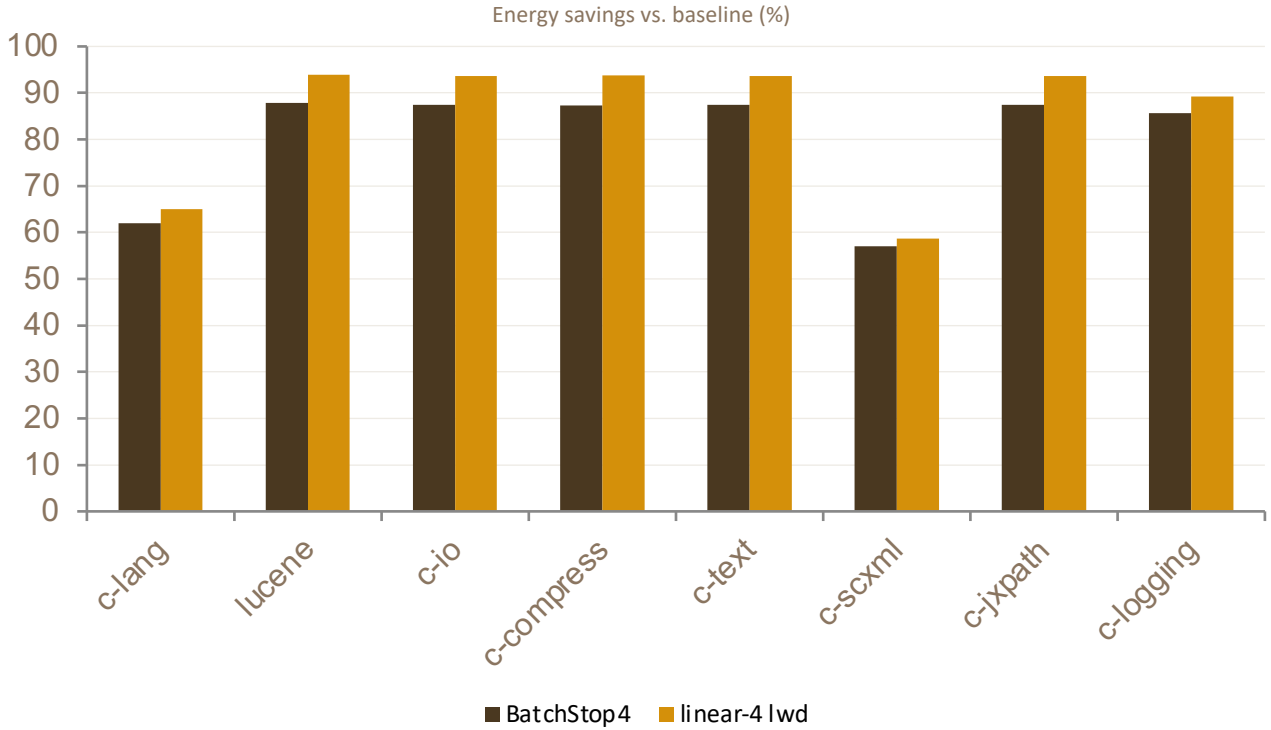
linear-4 lwd
Dynamic (adaptive size)

Both strategies cut energy by 57–94%



RQ1: ENERGY SAVINGS

Both strategies cut energy by 57–94%

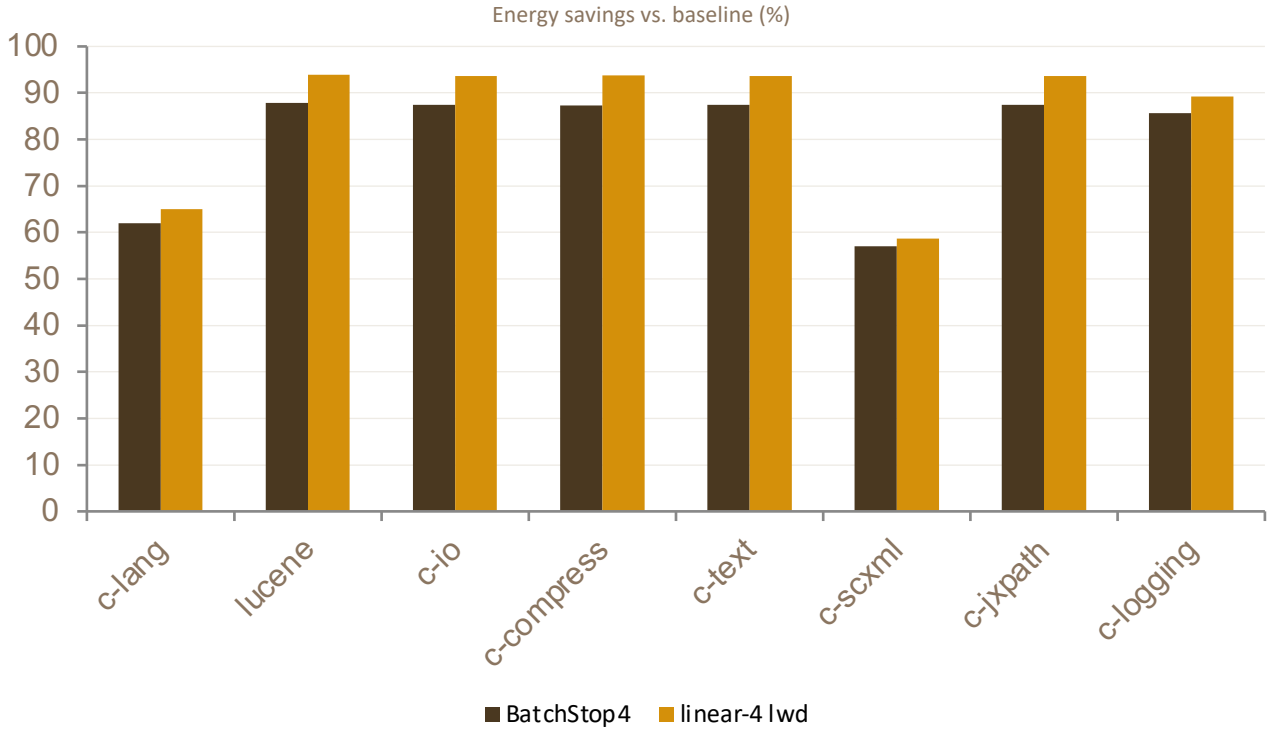


80.3%

BatchStop4
mean savings

RQ1: ENERGY SAVINGS

Both strategies cut energy by 57–94%



80.3%

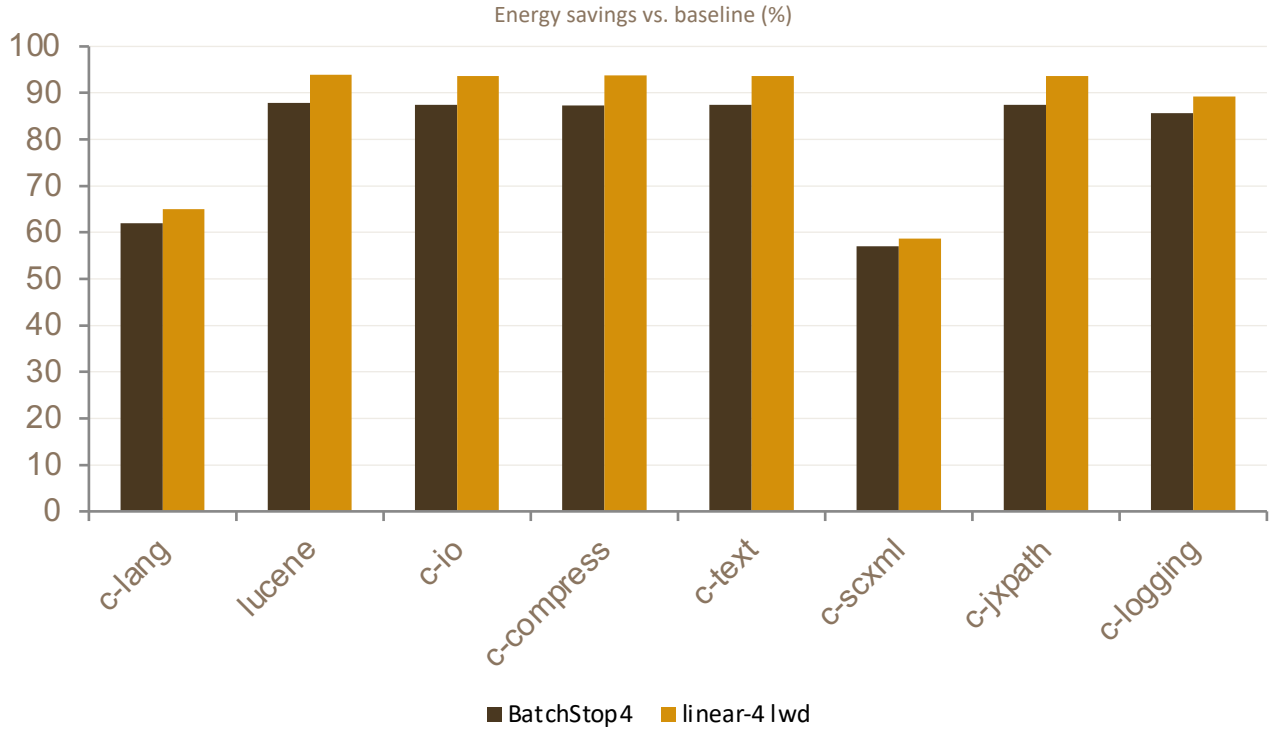
BatchStop4
mean savings

85.2%

linear-4 lwd
mean savings

RQ1: ENERGY SAVINGS

Both strategies cut energy by 57–94%



80.3%

BatchStop4
mean savings

85.2%

linear-4 lwd
mean savings

r >0.999

time–energy
correlation

RQ1: ENERGY SAVINGS

Absolute Energy Consumption (Wh)

commons-lang runs 10× longer than other projects, yet still saves 62–65%

Repository	Baseline (Wh)	BatchStop4 (Wh)	Save %	lin-4 lwd (Wh)	Save %
commons-lang	3029.00	1150.46	**62.0%**	1061.13	**65.0%**
lucene	375.86	45.92	**87.8%**	22.93	**93.9%**
commons-io	233.87	29.28	**87.5%**	14.62	**93.7%**
commons-compress	105.76	13.39	**87.3%**	6.60	**93.8%**
commons-text	58.67	7.42	**87.4%**	3.70	**93.7%**
commons-scxml	29.89	12.84	**57.0%**	12.34	**58.7%**
commons-jxpath	16.82	2.12	**87.4%**	1.06	**93.7%**
commons-logging	9.10	1.30	**85.7%**	0.99	**89.2%**

RQ1 · ENERGY SAVINGS, IN HUMAN TERMS

What a Year of Batch Testing Saves

~900
CI runs / year

1,968 Wh
saved per run

1,771 kWh
saved / year

~9,840 km
of EV driving

Assumptions: ~900 CI runs/year estimated from GitHub Actions run history (apache/commons-lang, "Java CI" workflow); savings from BatchStop4 vs. lin-4 lwd energy measurements; EV consumption at 180 Wh/km (typical mid-size EV average).

RQ1 · ENERGY SAVINGS, IN HUMAN TERMS

What a Year of Batch Testing Saves

~900
CI runs / year

1,968 Wh
saved per run

1,771 kWh
saved / year

~9,840 km
of EV driving

AMSTERDAM 🇳🇱

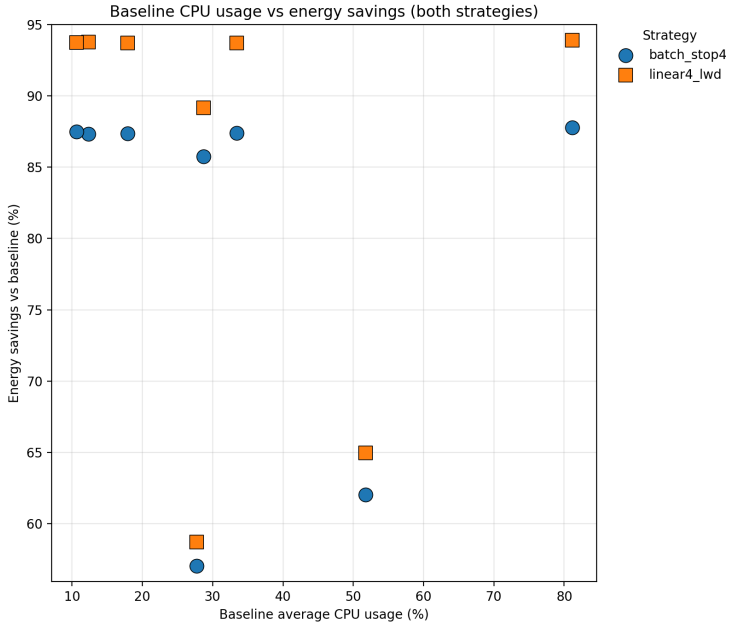
≈ 9,700–10,090 km by road

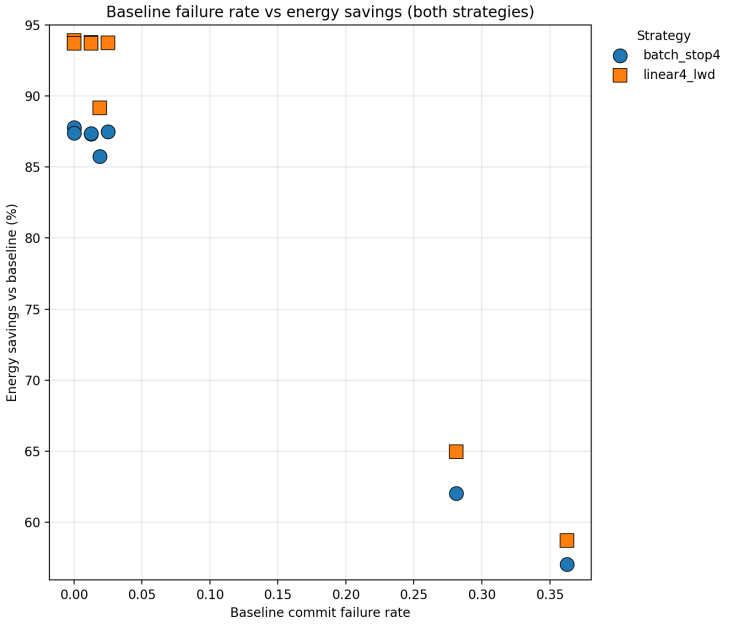
BEIJING 🇨🇳

Assumptions: ~900 CI runs/year estimated from GitHub Actions run history (apache/commons-lang, "Java CI" workflow); savings from BatchStop4 vs. lin-4 lwd energy measurements; EV consumption at 180 Wh/km (typical mid-size EV average).

RQ2: WHAT DRIVES THE SAVINGS?

Failure rate matters. CPU load does not.





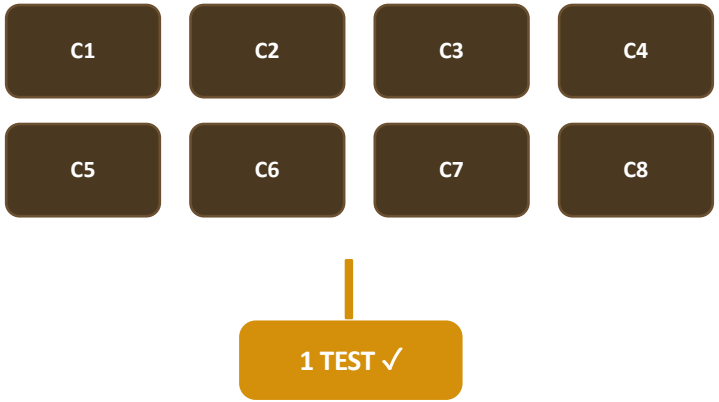
r = −0.14 p = 0.745 → NO correlation

r = −0.997 p < 0.0001 → STRONG correlation

RQ2: INTUITION

Why Failure Rate Determines Savings

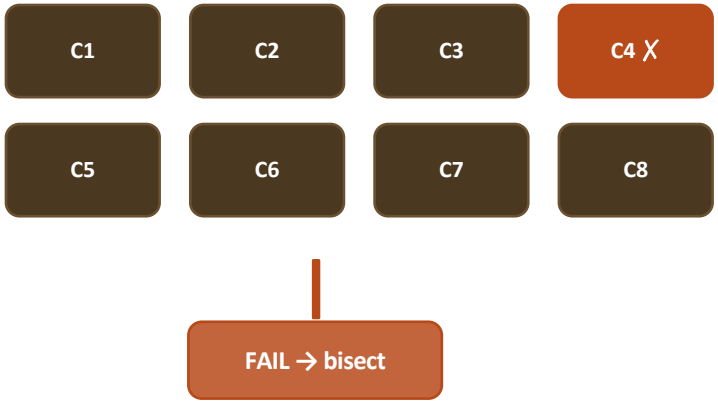
Low Failure Rate (e.g. lucene)



Saves 7 runs ✓

→ 87–94% energy saved

High Failure Rate (commons-scxml: 36%)



Extra re-runs required ✗

→ only 57–59% energy saved

PRACTICAL IMPLICATIONS

What should teams do?

A · Precondition

Fix instability first

1st step

before adopting batching

B · Algorithm choice

Simple batching is enough

>85%

savings, static algorithm

C · Break-even point

Worthwhile up to ~36% failure

57–59%

saved even at 36% failure

LIMITATIONS

Where This Study Falls Short

We only tested Java

Other languages or build tools could see different energy patterns.

One machine, not the cloud

Real cloud CI shares resources with other jobs, which could change the numbers.

Only CPU energy counted

Memory, disk, and network energy are real costs we did not capture.

We replayed old commits

We can’t know if developers would commit differently if batching were live.

FUTURE WORK

Open Directions

1. **Full-system energy:** Include DRAM, storage, network: Does batching benefit I/O-heavy suites differently from CPU-heavy ones?

2. **Beyond Java:** Python, JavaScript, C++ projects across diverse build systems and CI configurations.

3. **Cloud CI:** Shared infrastructure (e.g, AWS, GitHub Actions, Azure) adds real-world scheduling and resource variability.

4. **Live deployment:** Longitudinal study on active projects: How does batching change commit behaviour and team culture over time?

Replication package: doi:10.5281/zenodo.18507066

Key Takeaways

1. **Batch testing saves 57–94% energy across 8 real Java CI projects**
2. Energy savings ≈ time savings ($r > 0.999$). Batching shortens execution, not power draw
3. Failure rate is the dominant predictor. Lower failure rate → higher savings
4. CPU utilisation has NO significant effect on energy savings
5. **Simple static batching (BatchStop4) is highly effective at low failure rates**

Before deploying batch testing: stabilise your test suite first

Thank You

The Energy Impact of Batch Testing in Continuous Integration

QUESTION	Does batch testing save real energy, not just simulated executions?
METHOD	8 Java projects · 160 commits · 3 repeats · EnergiBridge on bare metal
FINDING	57–94% energy saved. Driven by failure rate, not CPU utilisation
TAKEAWAY	Stabilise your test suite first, even simple batching does the rest

📦 **Replication package:** doi.org/10.5281/zenodo.18507066