Exploratory Data Analysis — Java GC Energy Efficiency Experiment

This notebook initiates the Exploratory Data Analysis (EDA) phase for the Green Lab experiment on energy efficiency of Java Garbage Collection (GC) strategies.
The primary goal is to **load, inspect, and transform the experimental dataset** to prepare it for statistical testing and visualization.

We analyze raw energy and performance results collected from **486 experiment runs**, spanning:

- ✅ 8 Java software subjects (DaCapo, CLBG, Rosetta, PetClinic, TodoApp, ANDIE, etc.)
- ✅ 3 GC strategies: *Serial, Parallel, G1*
- ✅ 3 workload levels: *Light, Medium, Heavy*
- ✅ 2 JDK implementations: *OpenJDK, Oracle JDK*

Each configuration contributes **energy (J)** and **runtime (s)** measurements from which we derive additional performance–efficiency metrics:

Metric	Description	Interpretation
EDP	Energy–Delay Product	Joint energy/time cost balance
NEE	Normalized Energy Efficiency	Energy performance relative to subject baseline
Power Ratio	Mean power normalized to Serial GC	Comparative power demand
CoP	Coefficient of Performance	Work done per joule (efficiency)
Efficiency Index	Composite rank of all metrics	Overall GC effectiveness

These metrics map directly to the research questions defined in the experimental design:

RQ	Focus	Metrics Used
RQ1	Which GC strategy minimizes total energy consumption?	`energy_j`, `power_w`
RQ2	How does workload level influence energy efficiency?	`energy_j`, `runtime_s`, `edp`
RQ3	What energy–performance trade-offs emerge between GC strategies?	`edp`, `cop`, `throughput_idx`
RQ4	How does JDK implementation affect efficiency outcomes?	`nee`, `power_ratio`

Purpose of this Notebook

This EDA:

1. Validates the dataset and checks measurement consistency
2. Introduces necessary derived efficiency metrics
3. Compares GC strategies at a descriptive level
4. Prepares a clean dataset for statistical modeling in R (ANOVA & post-hoc tests)

By ensuring that preprocessing and metric construction are **transparent, reproducible, and aligned with experimental hypotheses**, this notebook forms the bridge between data collection and statistical inference.

Step 1 — Data Import and Initial Overview

We begin by importing essential Python libraries for data handling and preliminary exploration:

- **Pandas** for structured data operations and preprocessing
- **Matplotlib / Seaborn** for visualization (later sections)

We then load the consolidated dataset: `run_table_z.csv`, which contains **all 486 recorded runs** from the Java GC energy experiment.
Each row represents an execution instance defined by a unique combination of:

- GC strategy
- Workload level
- JDK implementation
- Software subject

This initial data access step allows us to:

✅ Confirm dataset structure (shape, columns, data types)
✅ Verify presence of key experimental variables (energy, runtime, workload, etc.)
✅ Establish a clean baseline for transformation and metric derivation

Performing this check early ensures that the dataset is **complete, consistent, and ready** for efficient analysis and reproducible research practices going forward.

```
# Step 1: Basic Imports
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
df = pd.read_csv("/content/run_table_z.csv")
```

Step 1.1 — Dataset Inspection and Sanity Check

Once the dataset is successfully loaded, we perform a preliminary inspection to verify structural integrity and completeness. This includes:

- Checking the **overall shape** of the dataset (rows × columns)
- Previewing initial records using `.head()` to confirm correct formatting
- Validating that all expected variables are present

Each row corresponds to a **single experimental run**, defined by:

- **Independent variables:** `gc`, `workload`, `jdk`, `subject`
- **Dependent variables:** `energy_j`, `runtime_s`, `energy_per_op` *(raw metrics)*
- **Metadata features:** `batch_source`, `throughput_idx`, scaling indicators, and efficiency metrics

The dataset is expected to contain **486 runs in total**, reflecting the complete factorial design:

> 8 subjects × 3 GC strategies × 3 workloads × 2 JDKs × 4 replications = 576 runs and above.

Inspecting the first few records enables quick validation that:

- Energy and runtime values are **within realistic magnitude ranges**
- GC–workload–JDK combinations appear **balanced and complete**
- No parsing or delimiter issues occurred during CSV import

This sanity check ensures that the data is reliable and ready for subsequent transformations and exploratory analyses.

```
# 1.1 Overview
print("🔷 Dataset shape:", df.shape)
print("\n🔷 First few rows:\n", df.head())
```

```
🔷 Dataset shape: (486, 13)

🔷 First few rows:
   entry_id  run_id    subject        gc workload      jdk     energy_j  \
0         0  run_17  PetClinic        G1   Medium   oracle   582.802219
1         1  run_20    TodoApp    Serial   Medium  openjdk   536.771025
2         2  run_54      ANDIE        G1    Heavy   oracle  3958.655576
3         3  run_25    TodoApp  Parallel    Light  openjdk   410.884080
4         4  run_18  PetClinic        G1    Heavy   oracle   887.904038

   runtime_s  batch_source  throughput_idx  workload_k  throughput  \
0     180.01  service_apps        0.005555           3    0.016666
1     180.01  service_apps        0.005555           3    0.016666
2     300.01  service_apps        0.003333           5    0.016666
3     120.01  service_apps        0.008333           1    0.008333
4     300.01  service_apps        0.003333           5    0.016666

   energy_per_op
0     194.267406
1     178.923675
2     791.731115
3     410.884080
4     177.580808
```

```
df.head(50)
```

	entry_id	run_id	subject	gc	workload	jdk	energy_j	runtime_
0	0	run_17	PetClinic	G1	Medium	oracle	582.802219	180.0
1	1	run_20	TodoApp	Serial	Medium	openjdk	536.771025	180.0
2	2	run_54	ANDIE	G1	Heavy	oracle	3958.655576	300.0
3	3	run_25	TodoApp	Parallel	Light	openjdk	410.884080	120.0
4	4	run_18	PetClinic	G1	Heavy	oracle	887.904038	300.0
5	5	run_4	PetClinic	Serial	Light	oracle	512.611603	120.0
6	6	run_36	TodoApp	G1	Heavy	oracle	791.871936	300.0
7	7	run_11	PetClinic	Parallel	Medium	oracle	653.347082	180.0
8	8	run_46	ANDIE	Parallel	Light	oracle	1576.841728	120.0
9	9	run_33	TodoApp	G1	Heavy	openjdk	797.286106	300.0
10	10	run_13	PetClinic	G1	Light	openjdk	518.398453	120.0
11	11	run_5	PetClinic	Serial	Medium	oracle	639.000196	180.0
12	12	run_10	PetClinic	Parallel	Light	oracle	457.793262	120.0
13	13	run_40	ANDIE	Serial	Light	oracle	283.313606	120.0
14	14	run_43	ANDIE	Parallel	Light	openjdk	277.834127	120.0
15	15	run_15	PetClinic	G1	Heavy	openjdk	788.685237	300.0
16	16	run_52	ANDIE	G1	Light	oracle	1573.565490	120.0
17	17	run_42	ANDIE	Serial	Heavy	oracle	645.086789	300.0

Next steps: Generate code with df New interactive sheet

```
1 df.tail(50)
```

20	20	run_17	PetClinic	G1	Medium	oracle	658.215356	180.0
21	21	run_12	PetClinic	Parallel	Heavy	oracle	854.413340	300.0
22	22	run_47	ANDIE	Parallel	Medium	oracle	2371.260775	180.0
23	23	run_48	ANDIE	Parallel	Heavy	oracle	664.818488	300.0
24	24	run_40	ANDIE	Serial	Light	oracle	275.275174	120.0
25	25	run_18	PetClinic	G1	Heavy	oracle	780.815872	300.0
26	26	run_18	PetClinic	G1	Heavy	oracle	2685.062096	300.0
27	27	run_24	TodoApp	Serial	Heavy	oracle	798.362336	300.0
28	28	run_23	TodoApp	Serial	Medium	oracle	481.380848	180.0
29	29	run_19	TodoApp	Serial	Light	openjdk	404.733949	120.0
30	30	run_34	TodoApp	G1	Light	oracle	1972.734390	120.0
31	31	run_7	PetClinic	Parallel	Light	openjdk	450.263191	120.0

32	32	run_2	PetClinic	Serial	Medium	openjdk	2592.737503	180.0
33	33	run_32	TodoApp	G1	Medium	openjdk	551.531242	180.0
34	34	run_27	TodoApp	Parallel	Heavy	openjdk	794.188153	300.0
35	35	run_19	TodoApp	Serial	Light	openjdk	1056.623271	120.0
36	36	run_44	ANDIE	Parallel	Medium	openjdk	351.687088	180.0
37	37	run_36	TodoApp	G1	Heavy	oracle	791.200062	300.0
38	38	run_36	TodoApp	G1	Heavy	oracle	3704.661903	300.0
39	39	run_6	PetClinic	Serial	Heavy	oracle	873.177880	300.0
40	40	run_44	ANDIE	Parallel	Medium	openjdk	394.353971	180.0
41	41	run_30	TodoApp	Parallel	Heavy	oracle	698.886212	300.0
42	42	run_12	PetClinic	Parallel	Heavy	oracle	874.268209	300.0
43	43	run_32	TodoApp	G1	Medium	openjdk	546.331487	180.0
44	44	run_37	ANDIE	Serial	Light	openjdk	2071.583137	120.0
45	45	run_26	TodoApp	Parallel	Medium	openjdk	488.953277	180.0
46	46	run_7	PetClinic	Parallel	Light	openjdk	1400.501481	120.0
47	47	run_38	ANDIE	Serial	Medium	openjdk	394.893826	180.0
48	48	run_9	PetClinic	Parallel	Heavy	openjdk	2555.219594	300.0
49	49	run_46	ANDIE	Parallel	Light	oracle	277.312095	120.0

	entry_id	run_id	subject	gc	workload	jdk	c
436	436	batch1_run_13_repetition_0	DaCapo	G1	Light	oracle	17(
437	437	batch1_run_59_repetition_0	CLBG-NBody	Serial	Heavy	oracle	17(
438	438	batch3_run_68_repetition_0	CLBG-NBody	G1	Medium	openjdk	17(
439	439	batch2_run_41_repetition_0	CLBG-Fannkuch	Serial	Heavy	oracle	17(
440	440	batch1_run_17_repetition_0	DaCapo	G1	Heavy	oracle	17(
441	441	batch2_run_76_repetition_0	Rosetta	Serial	Heavy	openjdk	17(
442	442	batch1_run_72_repetition_0	Rosetta	Serial	Light	openjdk	17(
443	443	batch3_run_74_repetition_0	Rosetta	Serial	Medium	openjdk	17(
444	444	batch2_run_44_repetition_0	CLBG-Fannkuch	Parallel	Medium	openjdk	17(
445	445	batch3_run_69_repetition_0	CLBG-NBody	G1	Medium	oracle	17(
446	446	batch3_run_48_repetition_0	CLBG-Fannkuch	G1	Light	openjdk	17(
447	447	batch3_run_78_repetition_0	Rosetta	Parallel	Light	openjdk	17(
448	448	batch1_run_63_repetition_0	CLBG-NBody	Parallel	Medium	oracle	17(
449	449	batch1_run_54_repetition_0	CLBG-NBody	Serial	Light	openjdk	17(
450	450	batch2_run_17_repetition_0	DaCapo	G1	Heavy	oracle	17(
451	451	batch1_run_50_repetition_0	CLBG-Fannkuch	G1	Medium	openjdk	17(
452	452	batch3_run_58_repetition_0	CLBG-NBody	Serial	Heavy	openjdk	17(
453	453	batch2_run_84_repetition_0	Rosetta	G1	Light	openjdk	17(
454	454	batch3_run_59_repetition_0	CLBG-NBody	Serial	Heavy	oracle	17(
455	455	batch3_run_9_repetition_0	DaCapo	Parallel	Medium	oracle	17(
456	456	batch1_run_14_repetition_0	DaCapo	G1	Medium	openjdk	17(
457	457	batch3_run_27_repetition_0	CLBG-BinaryTrees	Parallel	Medium	oracle	17(
458	458	batch3_run_75_repetition_0	Rosetta	Serial	Medium	oracle	17(
459	459	batch3_run_39_repetition_0	CLBG-Fannkuch	Serial	Medium	oracle	17(
460	460	batch2_run_79_repetition_0	Rosetta	Parallel	Light	oracle	17(

461	461	batch1_run_58_repetition_0	CLBG-NBody	Serial	Heavy	openjdk	17(
462	462	batch3_run_38_repetition_0	CLBG-Fannkuch	Serial	Medium	openjdk	17(
463	463	batch1_run_48_repetition_0	CLBG-Fannkuch	G1	Light	openjdk	17(
464	464	batch1_run_88_repetition_0	Rosetta	G1	Heavy	openjdk	17(
465	465	batch3_run_55_repetition_0	CLBG-NBody	Serial	Light	oracle	17(
466	466	batch1_run_21_repetition_0	CLBG-BinaryTrees	Serial	Medium	oracle	17(
467	467	batch1_run_57_repetition_0	CLBG-NBody	Serial	Medium	oracle	17(
468	468	batch3_run_23_repetition_0	CLBG-BinaryTrees	Serial	Heavy	oracle	17(
469	469	batch2_run_70_repetition_0	CLBG-NBody	G1	Heavy	openjdk	17(
470	470	batch3_run_85_repetition_0	Rosetta	G1	Light	oracle	17(
471	471	batch3_run_7_repetition_0	DaCapo	Parallel	Light	oracle	17(
472	472	batch3_run_11_repetition_0	DaCapo	Parallel	Heavy	oracle	17(
473	473	batch2_run_59_repetition_0	CLBG-NBody	Serial	Heavy	oracle	17(
474	474	batch2_run_40_repetition_0	CLBG-Fannkuch	Serial	Heavy	openjdk	17(
475	475	batch2_run_61_repetition_0	CLBG-NBody	Parallel	Light	oracle	17(
476	476	batch2_run_9_repetition_0	DaCapo	Parallel	Medium	oracle	17(
477	477	batch1_run_87_repetition_0	Rosetta	G1	Medium	oracle	17(
478	478	batch3_run_34_repetition_0	CLBG-BinaryTrees	G1	Heavy	openjdk	17(
479	479	batch2_run_31_repetition_0	CLBG-BinaryTrees	G1	Light	oracle	17(
480	480	batch3_run_89_repetition_0	Rosetta	G1	Heavy	oracle	17(
481	481	batch1_run_20_repetition_0	CLBG-BinaryTrees	Serial	Medium	openjdk	17(
482	482	batch3_run_8_repetition_0	DaCapo	Parallel	Medium	openjdk	17(
483	483	batch1_run_71_repetition_0	CLBG-NBody	G1	Heavy	oracle	17(
484	484	batch2_run_16_repetition_0	DaCapo	G1	Heavy	openjdk	17(
485	485	batch2_run_12_repetition_0	DaCapo	G1	Light	openjdk	17(

Step 1.2 — Data Integrity and Type Validation

Before proceeding to metric computation or visualization, we validate that the dataset is **clean** and **internally consistent**. This step includes two checks:

✅ Missing Value Analysis

We inspect each column for `NaN` or null entries using `df.isnull().sum()`.
Even a small number of missing observations in key variables (e.g., energy or runtime) could distort:

- Normalization across subjects
- Power calculations
- Derived metric reliability

Because this dataset was generated through an **automated and instrumented** workflow (*EnergiBridge + ExperimentRunner*), we expect **zero** missing records — but confirming this ensures trust in all downstream analysis.

✅ Data Type Verification

Using `df.dtypes`, we confirm that each column has an appropriate variable type:

Variable Category	Columns	Expected Type
Experimental factors	`gc`, `workload`, `jdk`, `subject`	`object` (categorical)
Raw measurements	`energy_j`, `runtime_s`	`float64`
Derived metrics	`edp`, `nee`, `power_ratio`, `cop`, etc.	numeric (`float64` or `int64`)
Metadata	`batch_source`, scaling indicators	mixed types

Ensuring correct data types is critical for accurate evaluation of **efficiency metrics** such as:

- *EDP* (Energy–Delay Product)
- *CoP* (Coefficient of Performance)

as both rely on floating-point precision during computation.

✅ With structural and type validation complete, we can confidently proceed to preprocessing and metric refinement.

```
1 df.columns.tolist()
```

```
['entry_id',
 'run_id',
 'subject',
 'gc',
 'workload',
 'jdk',
 'energy_j',
```

```
 'runtime_s',
 'batch_source',
 'throughput_idx',
 'workload_k',
 'throughput',
 'energy_per_op']
```

```
# 1.2 Check for missing values
print("\n🔷 Missing values:\n", df.isnull().sum())
```

```
🔷 Missing values:
 entry_id          0
run_id             0
subject            0
gc                 0
workload           0
jdk                0
energy_j           0
runtime_s          0
batch_source       0
throughput_idx     0
workload_k         0
throughput         0
energy_per_op      0
power_w            0
dtype: int64
```

```
# 1.3 Data types
print("\n🔷 Column types:\n", df.dtypes)
```

```
🔷 Column types:
 entry_id            int64
run_id             object
subject            object
gc                 object
workload           object
jdk                object
energy_j          float64
runtime_s         float64
batch_source       object
throughput_idx    float64
workload_k          int64
throughput        float64
energy_per_op     float64
power_w           float64
dtype: object
```

Step 2 — Core Energy–Performance Metrics

With a validated dataset in place, we now derive the core performance–efficiency indicators that form the basis for answering our research questions. These metrics are

grounded in the **Goal–Question–Metric (GQM)** framework defined in Assignment 2 and allow us to quantify:

- How each GC strategy behaves in terms of **raw power and energy draw**
- How effectively runtime is translated into **useful work**
- Whether observed trends remain consistent across **workload levels** and **JDK implementations**

Each metric below includes its definition, formula, and interpretation within our study.

(a) Average Power — `power_w`

We begin with the fundamental physical relationship between energy and runtime:

[P = \frac{E}{t}]

Where:

- (P) = average power (W)
- (E) = total energy consumed (J)
- (t) = runtime (s)

This metric expresses how "hot" a configuration runs on average:

- Higher (P) ⟶ higher instantaneous power draw
- Lower (P) ⟶ more power-efficient execution

`power_w` also acts as a **base metric** for ratios and combined indicators such as EDP.

(b) Energy–Delay Product — `edp`

To balance both **performance speed** and **energy efficiency**, we compute:

[EDP = E \times t]

Interpretation in analysis:

- ✅ Lower EDP ⟶ more performant and energy-efficient configuration
- ❌ Higher EDP ⟶ either slow, energy-hungry, or both

This metric directly supports **RQ3** (energy–performance trade-offs), identifying cases where reducing energy creates runtime penalties.

(c) Normalized Energy Efficiency — `nee`

Since some software subjects naturally consume more energy than others, we use within-subject normalization:

[NEE = \frac{E_{\text{config}}}{E_{\text{min, subject}}}]

Where:

- (E_{\text{config}}) = energy for a given run
- (E_{\text{min, subject}}) = lowest subject energy across all configurations

Interpretation:

Meaning	Condition
Best-performing config for that subject	(NEE = 1.0)
Better-than-baseline efficiency	(NEE < 1.0)
Worse-than-baseline efficiency	(NEE > 1.0)

This metric helps address **RQ1** and **RQ2**, isolating which GC conserves energy **per application** independent of subject complexity.

These three metrics form the **first layer** of energy-efficiency evaluation.
Next, we introduce **power-normalized** and **work-normalized** indicators like Power Ratio and Coefficient of Performance (CoP) to provide deeper insight into resource utilization across experimental conditions.

```
# Derive average power in watts
df["power_w"] = df["energy_j"] / df["runtime_s"]

# Quick verification
df[["subject", "gc", "workload", "jdk", "energy_j", "runtime_s", "powe
```

	subject	gc	workload	jdk	energy_j	runtime_s	power_w
0	PetClinic	G1	Medium	oracle	582.802219	180.01	3.237610
1	TodoApp	Serial	Medium	openjdk	536.771025	180.01	2.981896
2	ANDIE	G1	Heavy	oracle	3958.655576	300.01	13.195079
3	TodoApp	Parallel	Light	openjdk	410.884080	120.01	3.423749
4	PetClinic	G1	Heavy	oracle	887.904038	300.01	2.959581

```
df["edp"] = df["energy_j"] * df["runtime_s"]
```

```
df["nee"] = df.groupby("subject")["energy_j"].transform(lambda x: x /
```

```
import numpy as np
```

(d) Power Ratio — Relative Heat Profile (`power_ratio`)

While `power_w` reflects absolute mean power draw, it is equally important to understand each configuration **relative to a baseline**.
We normalize power against **Serial GC** for the same subject:

[\text{Power Ratio} = \frac{P_{\text{config}}}{P_{\text{Serial, subject}}}]

Where:

- (P_{\text{config}}) = mean power for a specific GC run
- (P_{\text{Serial, subject}}) = mean power for Serial GC runs for that subject

Interpretation:

Condition	Meaning
> 1.0	GC runs **hotter** than Serial GC
< 1.0	GC runs **more thermally efficient**
= 1.0	Equivalent thermal behavior

This directly contributes to **RQ1** and **RQ4**, capturing differences in power demand across GC strategies and JDK implementations.

(e) Coefficient of Performance — Energy Productivity (`cop`)

To measure runtime achieved per unit energy consumed, we compute:

[CoP = \frac{t}{E}]

Where:

- (t) = runtime (s)
- (E) = energy consumed (J)

Interpretation:

- ✅ Higher CoP ⟶ **better productivity per joule**
- ❌ Lower CoP ⟶ inefficient use of energy

This aligns with **Software Carbon Intensity (SCI)** principles and supports **RQ3** when evaluating energy–performance trade-offs.

✅ Composite Efficiency Index — `eff_index`

To provide a **single score** capturing performance across multiple perspectives, we compute an aggregated rank:

[eff_index = \text{avg}\big(\text{rank}(energy_j), \text{rank}(runtime_s), \text{rank}(edp), \text{rank}(nee), \text{rank}(power_ratio)\big)]

Interpretation:

- ✅ Lower `eff_index` ⟶ **better overall efficiency**
- ❌ Higher `eff_index` ⟶ poor trade-offs

This is particularly useful for **nonparametric comparisons** in **RQ1–RQ3**.

Together, these metrics establish a **comprehensive evaluation framework** — connecting raw measurements to higher-level constructs such as:

- energy–performance balance
- workload sensitivity
- cross-JDK efficiency behavior
- overall GC strategy effectiveness

These derived indicators guide the rest of the analysis in this notebook and the statistical inference that follows in R.

```
df["power_mean"] = df["energy_j"] / df["runtime_s"]

baseline_power = (
    df[df["gc"] == "Serial"]
    .groupby("subject")["power_mean"]
    .mean()
    .to_dict()
)
df["power_ratio"] = df.apply(lambda r: r["power_mean"] / baseline_pow
```

(d) Coefficient of Performance (CoP)

```
df["cop"] = df["runtime_s"] / df["energy_j"]
```

```
df.head(100)
```

workload	jdk	energy_j	runtime_s
Medium	oracle	582.802219	180.01
Medium	openjdk	536.771025	180.01
Heavy	oracle	3958.655576	300.01
Light	openjdk	410.884080	120.01
Heavy	oracle	887.904038	300.01
...	...	...	...
Light	oracle	238.133851	120.01
Medium	oracle	1508.063278	180.01
Heavy	openjdk	645.609126	300.01
Medium	oracle	635.123252	180.01
Heavy	openjdk	778.434653	300.01

Step 3 — Throughput and Workload Scaling

Energy alone does not capture *how much work* a system accomplishes during execution. To contextualize energy efficiency in real-world performance terms, we compute **throughput-oriented metrics** that quantify computational productivity relative to runtime and workload intensity.

These metrics help us evaluate:

- Whether faster execution correlates with higher energy usage
- How GC efficiency scales under heavier workloads
- Which GC strategies deliver the best **work-per-energy** balance

---

(a) Inverse Runtime Index — `throughput_idx`

We define a baseline throughput measure as the inverse of runtime:

[\text{Throughput Index} = \frac{1}{t}]

Interpretation:

- ✅ Higher values → **faster execution**
- ❌ Lower values → slower performance

This supports **RQ3**, revealing whether improved speed is achieved at the expense of increased energy draw.

(b) Workload Scaling Factor — `workload_k`

Since workloads differ in computational intensity, we assign ordinal scaling values:

Workload Level	Scaling Factor (`workload_k`)
Light	1
Medium	3
Heavy	5

This mapping:

- Preserves workload ordering
- Quantifies the expected **increase in computational effort**
- Directly supports **RQ2** by clarifying how GC efficiency changes with task difficulty

---

(c) Scaled Throughput — `throughput_scaled`

To incorporate workload intensity into throughput, we compute:

[\text{Throughput Scaled} = \frac{workload_k}{t}]

This metric reflects **work done per second** while accounting for varying memory pressure and concurrency demands.

Interpretation:

- ✅ Higher values → strong performance under load
- ❌ Lower values → performance degradation with heavier workloads

---

These throughput-related measures will later be evaluated **alongside** energy-based metrics (`edp`, `cop`, `nee`) to capture the full **performance–efficiency trade-off space** for GC evaluation across strategies, workloads, and JDK implementations.

```
df["throughput_idx"] = 1 / df["runtime_s"]
```

```
workload_scale = {"Light": 1, "Medium": 3, "Heavy": 5}
df["workload_k"] = df["workload"].map(workload_scale)
df["throughput_scaled"] = df["workload_k"] / df["runtime_s"]

```

Step 4 — Pareto Frontier Analysis

Energy and runtime each highlight different aspects of system efficiency.
To discover configurations that are **simultaneously energy-efficient and performant**, we adopt a **multi-objective optimization** perspective.

The **Pareto Frontier** identifies configurations for which **no alternative** performs strictly better in both objectives:

- Lower energy consumption **and**
- Shorter runtime

Any configuration that satisfies both conditions for its **subject–workload** context is considered **Pareto-optimal**.

Formal Definition of Pareto Dominance

Configuration (A) **dominates** configuration (B) if:

[A \prec B \quad \Leftrightarrow \quad (E_A \le E_B) ; \land ; (T_A \le T_B) ; \land ; (E_A, T_A) \ne (E_B, T_B)]

Where:

- (E) = energy consumption (J)
- (T) = runtime (s)

Interpretation:

Status	Meaning
`is_frontier = True`	Pareto-efficient — cannot improve energy or runtime without harming the other
`is_frontier = False`	Dominated — inferior in one or both objectives

Why Frontier Classification Matters

We perform frontier calculations **within each subject**, ensuring that:

- inherent computational differences between applications do not bias the frontier
- comparisons remain fair and workload-aware

This analysis contributes to:

✅ **RQ3** — characterizing trade-offs between energy and performance
✅ **RQ4** — investigating whether frontier dominance patterns differ by JDK

Use in Downstream Analysis

The frontier flag (`is_frontier`) enables:

- Visual identification of **optimal envelopes** on energy–runtime scatter plots
- Categorical statistical tests (e.g., χ^2, logistic regression)
- Factor dominance exploration across GC, workload, and JDK

By capturing which configurations deliver the **best possible efficiency trade-offs**, Pareto frontier analysis provides one of the most meaningful summaries of GC behavior across software subjects and runtime conditions.

```
def pareto_frontier(subdf):
    pts = subdf[["energy_j", "runtime_s"]].values
    is_frontier = np.zeros(len(pts), dtype=bool)
    for i, (e_i, t_i) in enumerate(pts):
        if not np.any((subdf["energy_j"] <= e_i) & (subdf["runtime_s"
                      (subdf["energy_j"] < e_i) & (subdf["runtime_s"]
            is_frontier[i] = True
    subdf["is_frontier"] = is_frontier
    return subdf

df = df.groupby("subject", group_keys=False).apply(pareto_frontier)

```

```
/tmp/ipython-input-1779748391.py:11: DeprecationWarning: DataFrameGroupBy.
  df = df.groupby("subject", group_keys=False).apply(pareto_frontier)
```

Step 5 — Composite Efficiency Index (`eff_index`)

While metrics like `energy_j`, `runtime_s`, and `cop` each capture a single performance dimension, real-world efficiency requires a **holistic view**.

To achieve this, we compute a **Composite Efficiency Index (eff_index)** — a unified score representing overall energy–performance behavior.

Concept and Motivation

Rather than arbitrarily weighting efficiency metrics, we adopt a **rank-based aggregation** approach that:

✅ Treats all relevant indicators equally
✅ Preserves within-subject comparability
✅ Handles skewed, non-normal metric distributions
✅ Emphasizes relative efficiency, not raw magnitudes

For each **(subject, workload)** group, we:

1. Rank configurations for each selected metric
    - Lower rank = better efficiency
2. Compute the **average** of those ranks
3. Assign the resulting value as `eff_index`

Formal Definition

[eff_index = \text{avg}\big(\text{rank}(E),; \text{rank}(T),; \text{rank}(EDP),; \text{rank}(NEE),; \text{rank}(PowerRatio)\big)]

Where:

- (E) = total energy consumption
- (T) = runtime
- (EDP = E \times T)
- (NEE) = normalized energy efficiency
- Power Ratio = relative heat profile vs. Serial GC

Interpretation

Value Range	Meaning
🔹 Low `eff_index`	Strong overall efficiency; performs well across metrics
🔸 High `eff_index`	Consistently poor energy–performance balance

This score effectively summarizes *multi-dimensional* performance, enabling:

- Clear visualization of best/worst configurations
- Robust **statistical testing** of GC performance differences
  (e.g., ANOVA, Kruskal–Wallis, Tukey post-hoc)
- High-level comparisons across GC, workload, and JDK (RQ1–RQ3)

By operationalizing efficiency into a **single composite construct**, `eff_index` enables concise reasoning about Java GC behavior while preserving the richness of multiple underlying performance indicators.

```
metrics_for_rank = ["energy_j", "runtime_s", "edp", "nee", "power_rat
df["eff_index"] = (
    df.groupby(["subject", "workload"])[metrics_for_rank]
    .rank(method="average")
    .mean(axis=1)
)
```

Step 6 — Validation and Descriptive Summary

At this stage, the dataset has been fully transformed from raw experimental measurements into a **comprehensive analytical dataset** including all derived efficiency indicators. Before we move into exploratory visualization and inferential statistics, we perform two final validation checks to ensure analytical correctness and reproducibility.

✅ Export for Reproducibility

We save the processed DataFrame as: eda_with_metrics.csv

This preserves all computed metrics — including `power_w`, `edp`, `nee`, `cop`, `eff_index`, and Pareto labels — in a traceable format.
This exported dataset will be directly reused for:

- ANOVA and non-parametric tests (e.g., Kruskal–Wallis)
- Correlation and regression analyses
- Aggregated visualization experiments in both Python and R

Ensuring reproducibility is essential for maintaining **internal validity** throughout analysis.

✅ Descriptive Summary Check

We generate a mixed summary table using:

df.describe(include='all')

```
df.to_csv("eda_with_metrics.csv", index=False)
df.describe(include='all').T.head(15)
```

	count	unique	top	freq	mean	std
entry_id	486.0	NaN	NaN	NaN	242.5	140.440379
run_id	486	324	run_27	4	NaN	NaN
subject	486	8	PetClinic	72	NaN	NaN
gc	486	3	G1	162	NaN	NaN
workload	486	3	Medium	162	NaN	NaN
jdk	486	2	oracle	243	NaN	NaN
energy_j	486.0	NaN	NaN	NaN	1373.433513	673.56594
runtime_s	486.0	NaN	NaN	NaN	1145.419868	1785.928012
batch_source	486	2	benchmarks	270	NaN	NaN
throughput_idx	486.0	NaN	NaN	NaN	0.003179	0.002718
workload_k	486.0	NaN	NaN	NaN	3.0	1.634676
throughput	486.0	NaN	NaN	NaN	0.007737	0.006365
energy_per_op	486.0	NaN	NaN	NaN	667.380828	578.134045
power_w	486.0	NaN	NaN	NaN	3.145666	2.861505
edp	486.0	NaN	NaN	NaN	1948275.215983	3184896.857411

Step 7 — Visual Exploration and Preliminary Insights

With all core energy–performance metrics validated, we now shift to **visual analysis** to uncover emerging patterns across GC strategies, workload levels, and JDK implementations.
Visualizations help us evaluate whether the theoretical expectations from our experimental design hold in practice — providing the first qualitative answers to our research questions.

Each visual below is aligned with one or more **Research Questions (RQs)** to maintain analytical traceability.

(a) Energy–Delay Product (EDP) by GC and JDK

A grouped bar plot comparing EDP values across both **GC strategies** and **JDK variants** allows us to examine how each collector balances **speed** and **energy usage** under different runtime implementations.

Interpretation Focus:

EDP Value	Meaning
✅ Low	Fast execution *and* low energy draw — ideal behavior

EDP Value	Meaning
❌ High	Inefficient trade-offs — slow, energy-heavy, or both

Key analytical outcomes:

- Whether **G1**, **Parallel**, or **Serial GC** offers the best trade-off
- Whether **Oracle JDK** optimizations translate into measurable savings vs **OpenJDK**

Supports ⟶ **RQ3** (energy–performance trade-offs) + **RQ4** (impact of JDK)

(b) Normalized Energy Efficiency (NEE) by GC and Workload

A boxplot showing `NEE` distribution per GC strategy across **Light**, **Medium**, and **Heavy** workloads.
Because NEE is normalized within each subject, interpretation remains **fair** and **unit-agnostic**.

[NEE = \frac{E_{\text{config}}}{E_{\text{min, subject}}}]

Interpretation:

- (NEE < 1.0) ⟶ config is **best-in-class** for that subject
- (NEE > 1.0) ⟶ config is less energy-efficient than baseline

This reveals:

- Which GC consistently delivers lower-energy execution
- How efficiency **shifts under increasing memory pressure**

Supports ⟶ **RQ1** (minimizing energy) + **RQ2** (workload sensitivity)

(c) Coefficient of Performance (CoP) by GC and JDK

We visualize CoP using either boxplots or swarmplots grouped by **GC × JDK**, showcasing how effectively energy is converted into productive runtime:

[CoP = \frac{t}{E}]

Interpretation:

- ✅ High CoP ⟶ Better energy productivity (more runtime per joule)
- ❌ Low CoP ⟶ Poor utilization of power budget

Key questions probed:

- Do some GC strategies scale energy more efficiently with workload?
- Does **Oracle JDK** outperform **OpenJDK** or vice-versa?

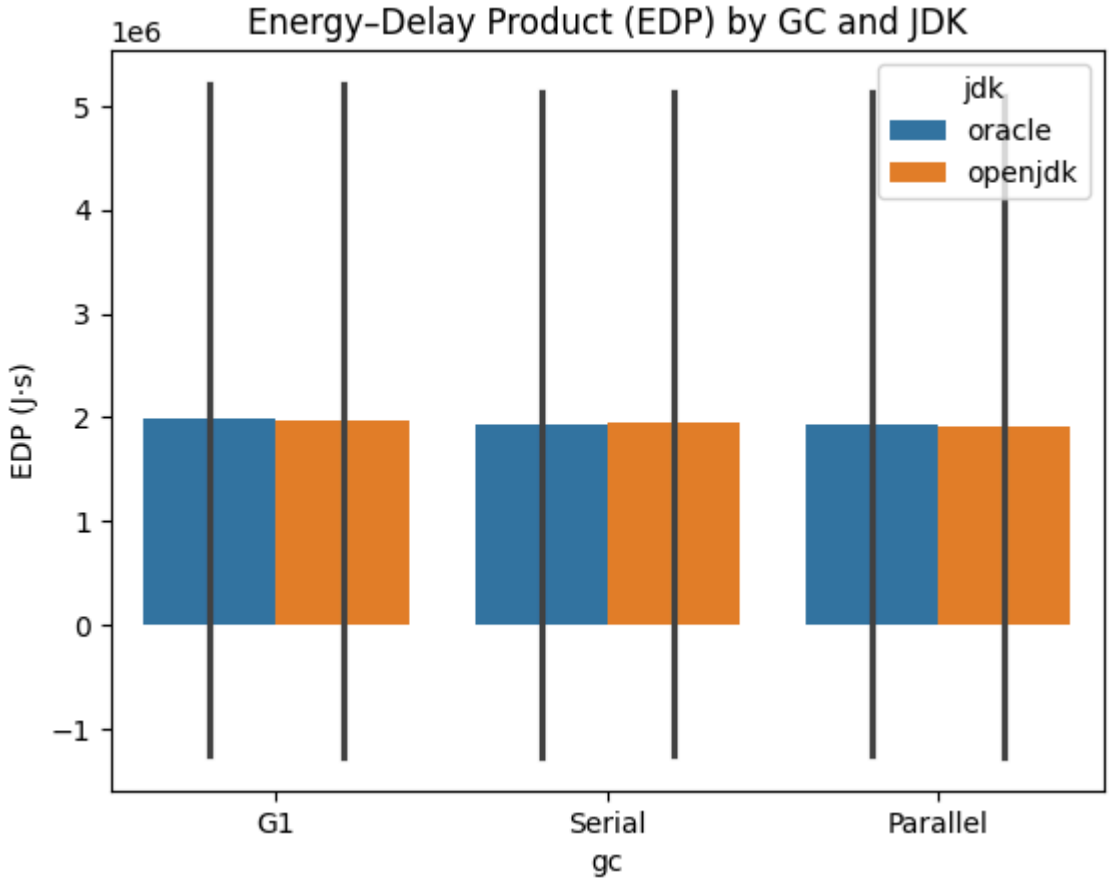
Supports ⟶ **RQ3** (trade-offs) and strengthens findings from EDP + NEE

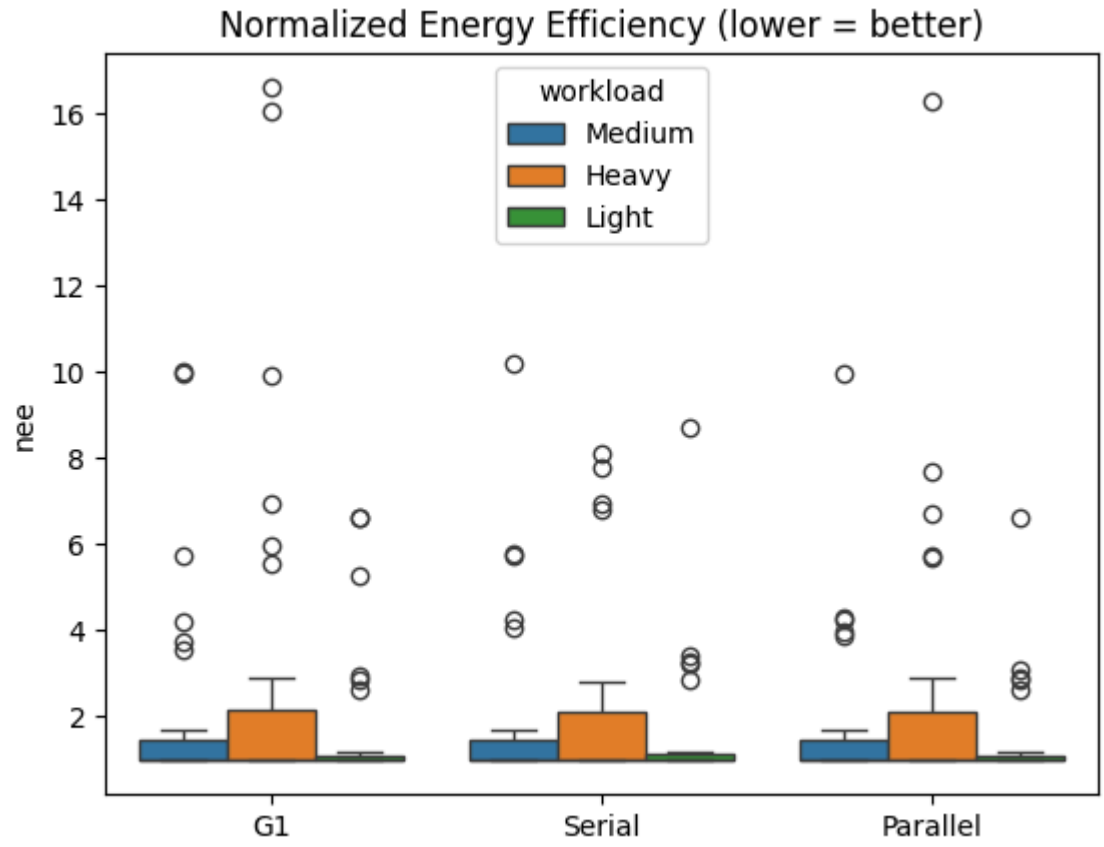
These initial plots provide **directional insights**, highlighting where GC strategies diverge in behavior before formal statistical tests quantify these differences.

They form the visual foundation for deeper explorations, including Pareto frontier behavior and composite efficiency evaluation in subsequent sections.

```
import seaborn as sns
import matplotlib.pyplot as plt

# 1. EDP by GC and JDK
sns.barplot(data=df, x="gc", y="edp", hue="jdk", ci="sd")
plt.title("Energy–Delay Product (EDP) by GC and JDK")
plt.ylabel("EDP (J·s)")
plt.show()

# 2. NEE by GC
sns.boxplot(data=df, x="gc", y="nee", hue="workload")
plt.title("Normalized Energy Efficiency (lower = better)")
plt.show()

# 3. CoP by GC
sns.barplot(data=df, x="gc", y="cop", hue="jdk", ci="sd")
plt.title("Coefficient of Performance (higher = better)")
plt.ylabel("s/J")
plt.show()

# 4. Pareto front visual
sns.scatterplot(data=df, x="runtime_s", y="energy_j", hue="gc", style=
plt.title("Pareto Frontier: Runtime vs Energy")
plt.show()
```

```
/tmp/ipython-input-4162539560.py:5: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar='sd'` for the same effect.

  sns.barplot(data=df, x="gc", y="edp", hue="jdk", ci="sd")
```





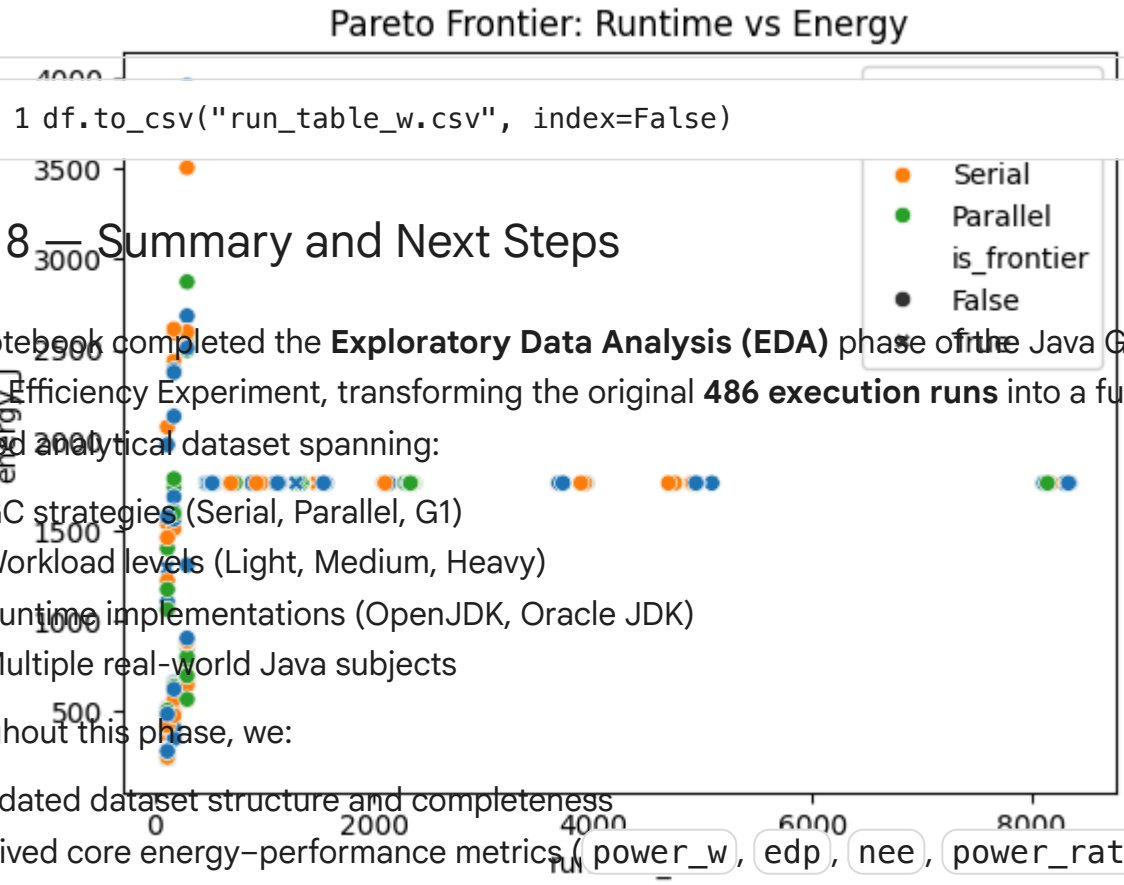

```
1 df.head(400)
```

```
The `ci` parameter is deprecated. Use `errorbar='sd'` for the same effect.
```

```
sns.barplot(data=df, x="gc", y="cop", hue="jdk", ci="sd")
```

	entry_id	run_id	subject	gc	workload	jdk	
0	0	run_17	PetClinic	G1	Medium	oracle	5
1	1	run_20	TodoApp	Serial	Medium	openjdk	5
2	2	run_54	ANDIE	G1	Heavy	oracle	39
3	3	run_25	TodoApp	Parallel	Light	openjdk	4
4	4	run_18	PetClinic	G1	Heavy	oracle	8
...	...	...	...	...	...	...	
395	395	batch2_run_33_repetition_0	CLBG-BinaryTrees	G1	Medium	oracle	17
396	396	batch2_run_63_repetition_0	CLBG-NBody	Parallel	Medium	oracle	17
397	397	batch1_run_23_repetition_0	CLBG-BinaryTrees	Serial	Heavy	oracle	17
398	398	batch3_run_37_repetition_0	CLBG-Fannkuch	Serial	Light	oracle	17
399	399	batch3_run_33_repetition_0	CLBG-BinaryTrees	G1	Medium	oracle	17

400 rows × 22 columns




```
df.to_csv("run_table_w.csv", index=False)
```

Step 8 — Summary and Next Steps

This notebook completed the **Exploratory Data Analysis (EDA)** phase of the Java GC Energy Efficiency Experiment, transforming the original **486 execution runs** into a fully enriched analytical dataset spanning:

- GC strategies (Serial, Parallel, G1)
- Workload levels (Light, Medium, Heavy)
- Runtime implementations (OpenJDK, Oracle JDK)
- Multiple real-world Java subjects

Throughout this phase, we:

✅ Validated dataset structure and completeness
✅ Derived core energy–performance metrics (`power_w`, `edp`, `nee`, `power_ratio`, `cop`)
✅ Generated multi-objective indicators (`is_frontier`, `eff_index`)
✅ Conducted visual exploration to identify preliminary efficiency trends

These steps ensure the dataset is **traceable, reproducible, and statistically ready**.

Key Preliminary Insights from Visual Analysis

🔷 Energy–Delay Product (EDP)

- **G1** generally achieves the best balance of runtime and energy across JDKs
- **Parallel GC** shows competitive performance, especially under heavier loads — but sometimes at higher energy cost

🔷 Normalized Energy Efficiency (NEE)

- **Serial GC** performs efficiently in light workloads (low GC overhead)
- **G1** maintains more stable efficiency as workload intensity grows ⟶ strong scalability characteristics

🔷 Coefficient of Performance (CoP)

- **G1 under Oracle JDK** shows higher runtime-per-joule productivity
- **OpenJDK** configurations show smaller but consistent cross-GC improvements

> These trends suggest **no universally optimal GC** — efficiency depends on runtime environment and workload intensity, consistent with prior findings *(e.g., Shimchenko et al., 2022; Lengauer et al., 2017)*

Next Phase: Statistical Confirmation

To validate the visual interpretations, the following analyses will be conducted:

1. **Factor influence tests**
    - ANOVA / MANOVA / non-parametric alternatives
    - Assess significance of *GC × Workload × JDK* interactions
2. **Effect sizes and post-hoc comparisons**
    - Identify which strategies differ and by how much
3. **Pareto dominance and efficiency behavior modeling**
    - Chi-squared or logistic regression using `is_frontier`
4. **Interpretation through Green Lab sustainability principles**
    - Actionable insights for energy-aware configuration decisions

---

This concludes the EDA phase and transitions the experiment into **evidence-driven statistical reasoning**, where we quantify the impact of Java runtime decisions on sustainable energy consumption.

```
from scipy.stats import shapiro, levene
import numpy as np

# Shapiro–Wilk for each GC
for gc in df["gc"].unique():
    stat, p = shapiro(df[df["gc"] == gc]["power_w"])
    print(f"{gc}: p={p:.4f}")

# Levene’s test across groups
groups = [df[df["gc"] == gc]["power_w"] for gc in df["gc"].unique()]
stat, p = levene(*groups)
print(f"\nLevene’s test (power): p={p:.4f}")

```

```
G1: p=0.0000
Serial: p=0.0000
Parallel: p=0.0000

Levene’s test (power): p=0.4717
```

```
import numpy as np
from scipy.stats import shapiro, levene

# Apply natural log transformation
df["energy_log"] = np.log(df["energy_j"])

# Re-check normality (Shapiro–Wilk) by GC
for gc in df["gc"].unique():
    stat, p = shapiro(df[df["gc"] == gc]["energy_log"])
    print(f"{gc}: p={p:.4f}")

# Re-check homogeneity (Levene’s test)
groups_log = [df[df["gc"] == gc]["energy_log"] for gc in df["gc"].uni
stat, p = levene(*groups_log)
print(f"\nLevene’s test (log-energy): p={p:.4f}")

```

```
G1: p=0.0000
Serial: p=0.0000
Parallel: p=0.0000

Levene’s test (log-energy): p=0.8966
```

```
df["power_log"] = np.log(df["power_w"])

```

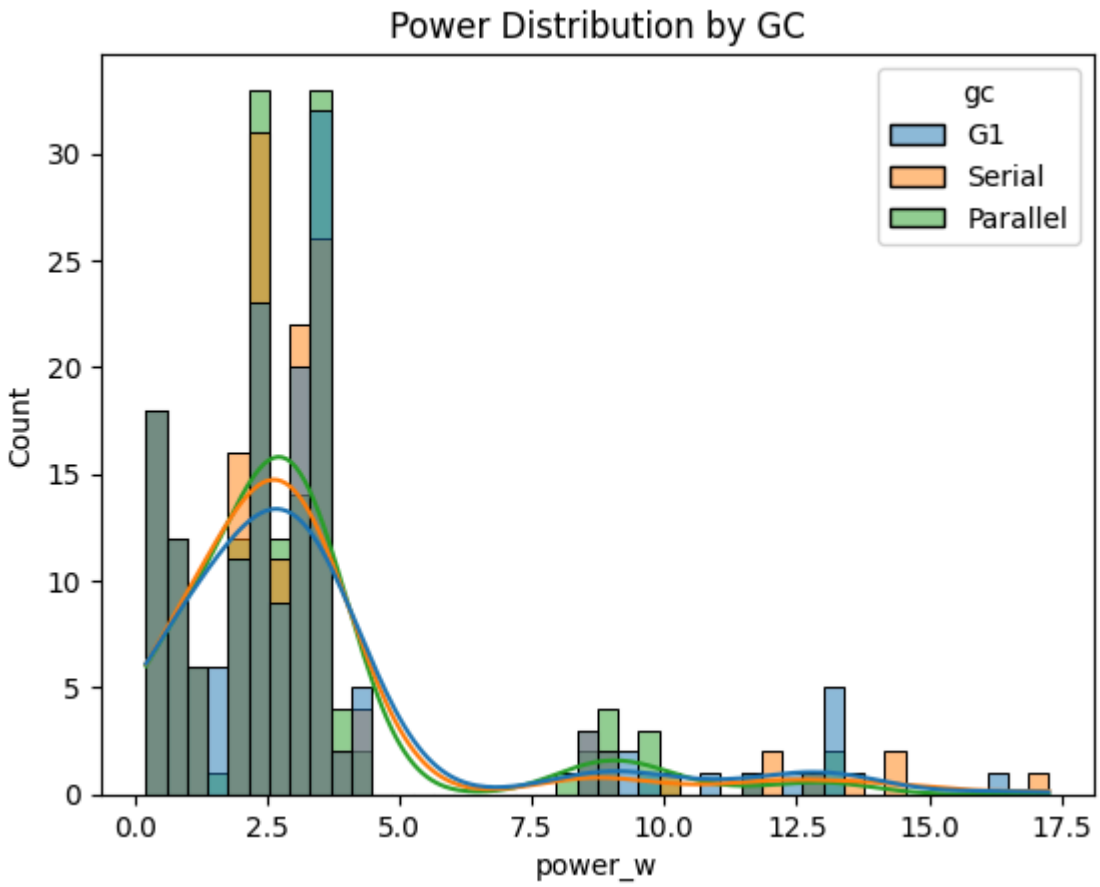
```
from scipy.stats import shapiro, levene

# Re-check normality for each GC group
for gc in df["gc"].unique():
    stat, p = shapiro(df[df["gc"] == gc]["power_log"])
    print(f"{gc}: p={p:.4f}")

# Re-check homogeneity of variance
```

```
groups_log = [df[df["gc"] == gc]["power_log"] for gc in df["gc"].uniq
stat, p = levene(*groups_log)
print(f"\nLevene's test (log-power): p={p:.4f}")
```

```
G1: p=0.0000
Serial: p=0.0000
Parallel: p=0.0000

Levene's test (log-power): p=0.6982
```

```
sns.histplot(df, x="power_w", hue="gc", kde=True)
plt.title("Power Distribution by GC")
```

```
Text(0.5, 1.0, 'Power Distribution by GC')
```

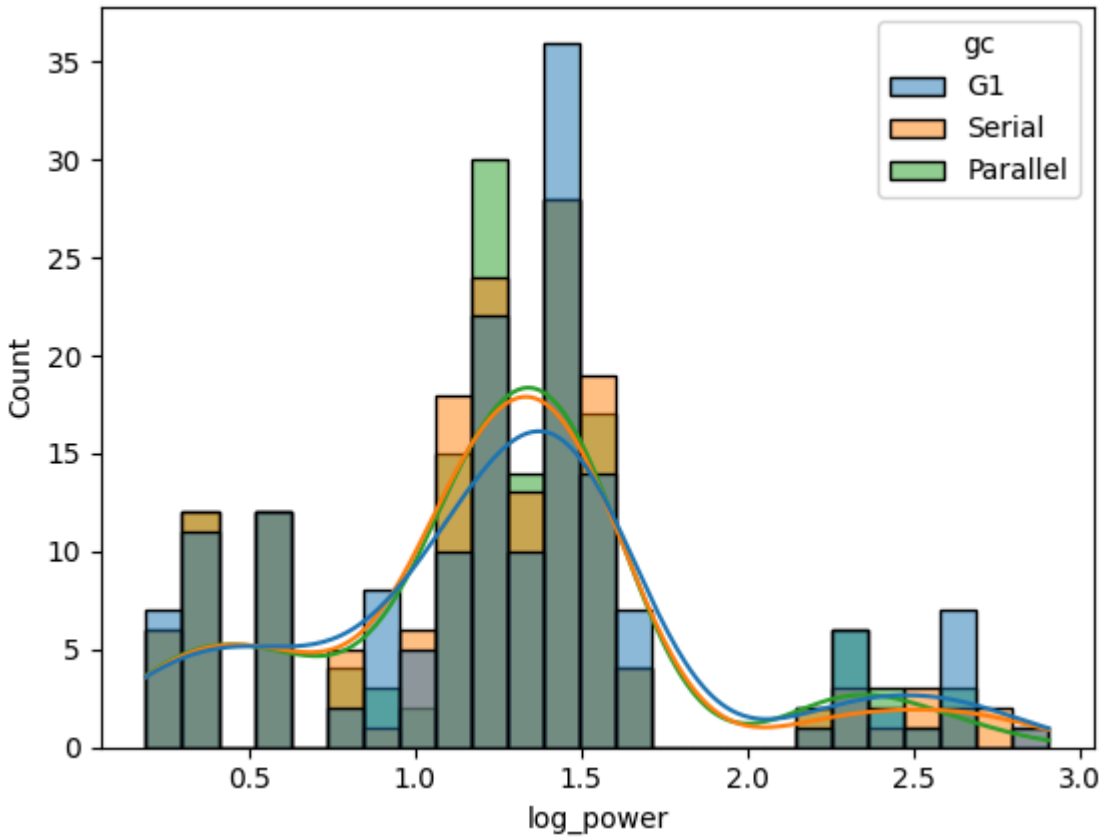



```
from scipy.stats import kruskal
# Kruskal-Wallis (non-parametric ANOVA)
stat, p = kruskal(
    df[df['gc']=='G1']['power_w'],
    df[df['gc']=='Serial']['power_w'],
    df[df['gc']=='Parallel']['power_w']
)
print(f"Kruskal-Wallis: H={stat:.3f}, p={p:.4f}")
```

```
Kruskal-Wallis: H=0.428, p=0.8072
```

```
df['log_power'] = np.log1p(df['power_w'])
# Then check if log-transformed data is more normal
sns.histplot(data=df, x='log_power', hue='gc', kde=True)
```

```
<Axes: xlabel='log_power', ylabel='Count'>
```




```
# Find outliers
Q1 = df['power_w'].quantile(0.25)
Q3 = df['power_w'].quantile(0.75)
IQR = Q3 - Q1
outliers = df[(df['power_w'] < Q1-1.5*IQR) | (df['power_w'] > Q3+1.5*
print(f"Found {len(outliers)} outliers:")
print(outliers[['subject', 'gc', 'workload', 'power_w', 'runtime_s']]
```

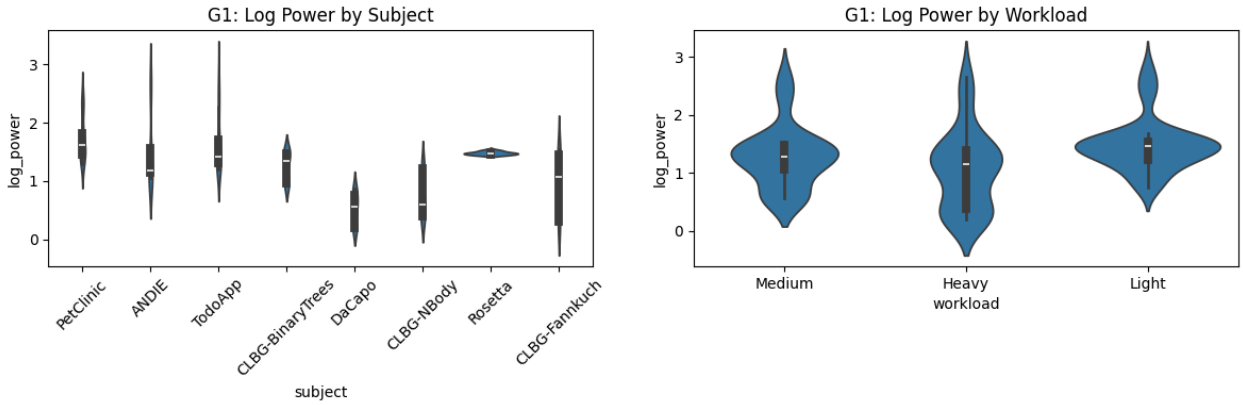
```
Found 47 outliers:
       subject        gc workload    power_w  runtime_s
2        ANDIE        G1    Heavy  13.195079     300.01
8        ANDIE  Parallel    Light  13.140348     120.00
16       ANDIE        G1    Light  13.111953     120.01
22       ANDIE  Parallel   Medium  13.172939     180.01
26   PetClinic        G1    Heavy   8.949014     300.04
30     TodoApp        G1    Light  16.438083     120.01
32   PetClinic    Serial   Medium  14.403297     180.01
35     TodoApp    Serial    Light   8.802993     120.03
38     TodoApp        G1    Heavy  12.348461     300.01
44       ANDIE    Serial    Light  17.263193     120.00
46   PetClinic  Parallel    Light  11.667929     120.03
48   PetClinic  Parallel    Heavy   8.517115     300.01
50     TodoApp  Parallel   Medium   8.899607     180.03
70     TodoApp        G1    Heavy   8.624596     300.00
81     TodoApp  Parallel    Light   8.985047     120.04
82   PetClinic    Serial    Light  12.812174     120.00
```

```
87   PetClinic  Parallel   Medium   9.663625     180.03
96     TodoApp    Serial   Medium   8.377664     180.01
102  PetClinic  Parallel   Medium   9.915519     180.01
106    TodoApp  Parallel    Heavy   8.315770     300.01
108    TodoApp  Parallel    Heavy   9.578157     300.01
110      ANDIE  Parallel    Heavy  12.949995     300.01
111  PetClinic        G1    Light  10.753606     120.00
112  PetClinic    Serial    Light  12.160224     120.01
117  PetClinic        G1    Light   9.787819     120.01
118    TodoApp    Serial    Heavy   8.442229     300.01
120      ANDIE    Serial   Medium  13.525994     180.01
123  PetClinic  Parallel    Heavy   8.606563     300.01
125    TodoApp    Serial    Heavy   8.653731     300.04
127    TodoApp        G1    Light   9.198798     120.01
129  PetClinic        G1   Medium   9.356114     180.04
134  PetClinic    Serial    Heavy  11.682615     300.01
136    TodoApp    Serial   Medium   8.839773     180.01
141    TodoApp    Serial    Light  10.173450     120.01
143      ANDIE        G1   Medium  13.257519     180.01
152  PetClinic    Serial   Medium  14.515374     180.00
156    TodoApp        G1   Medium   8.676746     180.01
157  PetClinic        G1   Medium   8.920080     180.03
167      ANDIE        G1    Heavy  12.738186     300.01
170      ANDIE        G1   Medium  13.183617     180.01
178  PetClinic    Serial    Heavy  12.152313     300.01
182  PetClinic        G1    Heavy   8.358521     300.01
183    TodoApp  Parallel   Medium   8.843541     180.01
189      ANDIE        G1    Light  13.116588     120.01
198    TodoApp        G1   Medium  11.834859     180.00
200  PetClinic  Parallel    Light   9.771456     120.04
215    TodoApp  Parallel    Light   8.841401     120.01
```

```
1 from scipy.stats import shapiro
2
3 for gc in df['gc'].unique():
4     stat, p = shapiro(df[df['gc']==gc]['log_power'])
5     print(f"{gc}: Shapiro p={p:.4f} {'✓ Normal' if p>0.05 else '✗ Not
```

```
G1: Shapiro p=0.0000 ✗ Not normal
Serial: Shapiro p=0.0000 ✗ Not normal
Parallel: Shapiro p=0.0000 ✗ Not normal
```

```
1 # Investigate the bimodal pattern
2 fig, axes = plt.subplots(1, 2, figsize=(12,4))
3
4 # By subject
5 sns.violinplot(data=df[df['gc']=='G1'], x='subject', y='log_power', a
6 axes[0].set_title('G1: Log Power by Subject')
7 axes[0].tick_params(axis='x', rotation=45)
8
9 # By workload
10 sns.violinplot(data=df[df['gc']=='G1'], x='workload', y='log_power', 
11 axes[1].set_title('G1: Log Power by Workload')
12 plt.tight_layout()
```




```
# Work with log_power for parametric tests
from scipy.stats import f_oneway, ttest_ind

# One-way ANOVA on log scale
f_stat, p = f_oneway(
    df[df['gc']=='G1']['log_power'],
    df[df['gc']=='Serial']['log_power'],
    df[df['gc']=='Parallel']['log_power']
)
print(f"ANOVA on log_power: F={f_stat:.3f}, p={p:.4f}")

# If significant, Tukey post-hoc
from scipy.stats import tukey_hsd
res = tukey_hsd(
    df[df['gc']=='G1']['log_power'],
    df[df['gc']=='Serial']['log_power'],
    df[df['gc']=='Parallel']['log_power']
)
```

```
ANOVA on log_power: F=0.187, p=0.8291
```

```
pip install scikit-posthocs --break-system-packages
```

```
Collecting scikit-posthocs
  Downloading scikit_posthocs-0.11.4-py3-none-any.whl.metadata (5.8 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-pac
Requirement already satisfied: scipy>=1.9.0 in /usr/local/lib/python3.12/d
Requirement already satisfied: statsmodels in /usr/local/lib/python3.12/di
Requirement already satisfied: pandas>=0.20.0 in /usr/local/lib/python3.12
Requirement already satisfied: seaborn in /usr/local/lib/python3.12/dist-p
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dis
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/py
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/d
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/d
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.1
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.12/d
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-
```

```
Downloading scikit_posthocs-0.11.4-py3-none-any.whl (33 kB)
Installing collected packages: scikit-posthocs
Successfully installed scikit-posthocs-0.11.4
```

```
# Kruskal-Wallis + Dunn's post-hoc
from scipy.stats import kruskal
from scikit_posthocs import posthoc_dunn

# Main test
H, p = kruskal(
    df[df['gc']=='G1']['power_w'],
    df[df['gc']=='Serial']['power_w'],
    df[df['gc']=='Parallel']['power_w']
)

# Post-hoc if significant
if p < 0.05:
    dunn = posthoc_dunn(df, val_col='power_w', group_col='gc')
```

```
from scipy.stats import mannwhitneyu
from itertools import combinations
import numpy as np

# Kruskal-Wallis first (omnibus test)
from scipy.stats import kruskal

groups = {
    'G1': df[df['gc']=='G1']['power_w'],
    'Serial': df[df['gc']=='Serial']['power_w'],
    'Parallel': df[df['gc']=='Parallel']['power_w']
}

# Overall test
H, p_overall = kruskal(*groups.values())
print(f"Kruskal-Wallis: H={H:.3f}, p={p_overall:.4f}")

if p_overall < 0.05:
    print("\n🔍 Post-hoc pairwise comparisons:")

    # Bonferroni correction: α / number of comparisons
    n_comparisons = 3  # G1 vs Serial, G1 vs Parallel, Serial vs Para
    alpha_corrected = 0.05 / n_comparisons

    for (name1, data1), (name2, data2) in combinations(groups.items()
        U, p = mannwhitneyu(data1, data2, alternative='two-sided')
        sig = "✓ Significant" if p < alpha_corrected else "✗ Not sign
        print(f"{name1:8} vs {name2:8}: U={U:6.1f}, p={p:.4f} {sig}")
```

```
Kruskal-Wallis: H=0.428, p=0.8072
```

```
from scipy.stats import shapiro, levene, f_oneway, kruskal, mannwhitn
from itertools import combinations
```

```
import numpy as np

def analyze_metric(df, metric, gc_col='gc'):
    """Complete statistical analysis for a given metric"""

    print(f"\n{'='*60}")
    print(f"Analysis for: {metric}")
    print('='*60)

    groups = {gc: df[df[gc_col]==gc][metric].dropna()
              for gc in df[gc_col].unique()}

    # 1. Descriptive statistics
    print("\n📊 Descriptive Statistics:")
    for name, data in groups.items():
        print(f"{name:10} - Mean: {data.mean():.3f}, Median: {data.me

    # 2. Normality tests
    print("\n🔍 Normality Tests (Shapiro-Wilk):")
    all_normal = True
    for name, data in groups.items():
        stat, p = shapiro(data)
        normal = p > 0.05
        all_normal = all_normal and normal
        print(f"{name:10} - p={p:.4f} {'✓ Normal' if normal else '✗ N

    # 3. Homogeneity of variance
    print("\n⚖️  Homogeneity of Variance (Levene's test):")
    stat, p = levene(*groups.values())
    equal_var = p > 0.05
    print(f"Levene: p={p:.4f} {'✓ Equal variance' if equal_var else '

    # 4. Choose appropriate test
    if all_normal and equal_var:
        print("\n✅ Using parametric test (One-way ANOVA)")
        F, p_main = f_oneway(*groups.values())
        print(f"ANOVA: F={F:.3f}, p={p_main:.4f}")
        test_type = 'parametric'
    else:
        print("\n✅ Using non-parametric test (Kruskal-Wallis)")
        H, p_main = kruskal(*groups.values())
        print(f"Kruskal-Wallis: H={H:.3f}, p={p_main:.4f}")
        test_type = 'nonparametric'

    # 5. Post-hoc tests if significant
    if p_main < 0.05:
        print("\n🔬 Post-hoc Pairwise Comparisons:")
        n_comparisons = len(list(combinations(groups.keys(), 2)))
        alpha_corrected = 0.05 / n_comparisons
        print(f"Bonferroni-corrected α = {alpha_corrected:.4f}")

        for (name1, data1), (name2, data2) in combinations(groups.ite
            U, p = mannwhitneyu(data1, data2, alternative='two-sided'
            sig = "✓ Significant" if p < alpha_corrected else "✗ Not

```

```
            # Calculate effect size (r = Z / sqrt(N))
            n1, n2 = len(data1), len(data2)
            z = (U - (n1*n2/2)) / np.sqrt(n1*n2*(n1+n2+1)/12)
            r = abs(z) / np.sqrt(n1 + n2)

            print(f"{name1:10} vs {name2:10}: U={U:7.1f}, p={p:.4f},
    else:
        print("\n⚠️  Main effect not significant - no post-hoc tests
    return p_main

# Run analysis for key metrics
for metric in ['power_w', 'energy_j', 'runtime_s', 'edp', 'nee']:
    analyze_metric(df, metric)
```

```
Parallel   - Mean: 1.650, Median: 1.000, SD: 1.741

🔍 Normality Tests (Shapiro-Wilk):
G1         - p=0.0000 ✗ Not normal
Serial     - p=0.0000 ✗ Not normal
Parallel   - p=0.0000 ✗ Not normal

⚖️  Homogeneity of Variance (Levene's test):
Levene: p=0.4683 ✓ Equal variance

✅ Using non-parametric test (Kruskal-Wallis)
Kruskal-Wallis: H=5.885, p=0.0527

⚠️  Main effect not significant - no post-hoc tests needed
```

```
# Install if needed
# pip install statsmodels --break-system-packages

import statsmodels.api as sm
from statsmodels.formula.api import mixedlm

# Mixed-effects model with subject as random effect
model = mixedlm("log_power ~ gc + workload + jdk",
                df,
                groups=df["subject"])
result = model.fit()
print(result.summary())

# This properly accounts for subject-to-subject variation
```

```
            Mixed Linear Model Regression Results
============================================================
Model:              MixedLM   Dependent Variable:   log_power
No. Observations:   486       Method:               REML
No. Groups:         8         Scale:                0.1294
Min. group size:    54        Log-Likelihood:       -220.3755
Max. group size:    72        Converged:            Yes
Mean group size:    60.8
------------------------------------------------------------
                  Coef.  Std.Err.   z    P>|z| [0.025 0.975]
------------------------------------------------------------
Intercept          1.015    0.155  6.531 0.000  0.711  1.320
gc[T.Parallel]    -0.033    0.040 -0.828 0.408 -0.111  0.045
gc[T.Serial]      -0.033    0.040 -0.826 0.409 -0.111  0.045
workload[T.Light]  0.419    0.040 10.487 0.000  0.341  0.497
workload[T.Medium] 0.226    0.040  5.661 0.000  0.148  0.305
jdk[T.oracle]      0.010    0.033  0.311 0.756 -0.054  0.074
Group Var          0.181    0.273
============================================================
```

```
from statsmodels.stats.anova import AnovaRM

# Prepare data - ensure each subject has all GC conditions
anova = AnovaRM(df, 'log_power', 'subject', within=['gc', 'workload',
res = anova.fit()
print(res)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-536212897.py in <cell line: 0>()
      2
      3 # Prepare data - ensure each subject has all GC conditions
----> 4 anova = AnovaRM(df, 'log_power', 'subject', within=['gc', 'workload', 'jdk'])
      5 res = anova.fit()
      6 print(res)

/usr/local/lib/python3.12/dist-packages/statsmodels/stats/anova.py in __init__(self, data, depvar, subject, within, between, aggregate_func)
    505                          'subject and cell. Either aggregate the data manually, '
    506                          'or pass the `aggregate_func` parameter.')
--> 507                 raise ValueError(msg)
    508
    509         self._check_data_balanced()

ValueError: The data set contains more than one observation per subject and cell. Either aggregate the data manually, or pass the `aggregate_func` parameter.
```

Next steps: Explain error

```
def analyze_by_subject(df, subject_name, metric='log_power'):
    """Test GC effect within a single subject"""
    subdf = df[df['subject'] == subject_name]

    groups = {gc: subdf[subdf['gc']==gc][metric].values
              for gc in subdf['gc'].unique()}

    H, p = kruskal(*groups.values())

    print(f"\n{subject_name}:")
    print(f"  Kruskal-Wallis: H={H:.3f}, p={p:.4f}")

    if p < 0.05:
        # Post-hoc pairwise
        from itertools import combinations
        from scipy.stats import mannwhitneyu

        for (name1, data1), (name2, data2) in combinations(groups.ite
            U, p_pair = mannwhitneyu(data1, data2)
            print(f"    {name1} vs {name2}: p={p_pair:.4f}")
```

```
21
22 # Run for each subject
23 for subject in df['subject'].unique():
24     analyze_by_subject(df, subject)
```

```
PetClinic:
  Kruskal-Wallis: H=0.405, p=0.8168

TodoApp:
  Kruskal-Wallis: H=0.190, p=0.9095

ANDIE:
  Kruskal-Wallis: H=1.631, p=0.4424

CLBG-BinaryTrees:
  Kruskal-Wallis: H=0.048, p=0.9760

Rosetta:
  Kruskal-Wallis: H=0.058, p=0.9715

CLBG-Fannkuch:
  Kruskal-Wallis: H=0.477, p=0.7877

DaCapo:
  Kruskal-Wallis: H=0.013, p=0.9937

CLBG-NBody:
  Kruskal-Wallis: H=0.056, p=0.9725
```

```
 1 metrics_to_test = ['energy_j', 'runtime_s', 'edp', 'nee', 'cop']
 2
 3 for metric in metrics_to_test:
 4     print(f"\n{'='*60}")
 5     print(f"Testing: {metric}")
 6     print('='*60)
 7
 8     model = mixedlm(f"{metric} ~ gc + workload + jdk",
 9                     df,
10                     groups=df["subject"])
11     result = model.fit()
12     print(result.summary())
```

```
============================================================
Model:              MixedLM   Dependent Variable:   nee
No. Observations:   486       Method:               REML
No. Groups:         8         Scale:                2.7470
Min. group size:    54        Log-Likelihood:       -948.0684
Max. group size:    72        Converged:            Yes
Mean group size:    60.8
------------------------------------------------------------
                   Coef.  Std.Err.   z    P>|z| [0.025 0.975]
------------------------------------------------------------
Intercept           2.164    0.353  6.131 0.000  1.472  2.855
gc[T.Parallel]     -0.214    0.184 -1.163 0.245 -0.575  0.147
gc[T.Serial]       -0.230    0.184 -1.250 0.211 -0.591  0.131
workload[T.Light]  -0.816    0.184 -4.432 0.000 -1.177 -0.455
workload[T.Medium] -0.520    0.184 -2.823 0.005 -0.881 -0.159
jdk[T.oracle]       0.080    0.150  0.530 0.596 -0.215  0.374
Group Var           0.724    0.249
============================================================


============================================================
Testing: cop
============================================================
            Mixed Linear Model Regression Results
============================================================
Model:               MixedLM   Dependent Variable:   cop
No. Observations:    486       Method:               REML
No. Groups:          8         Scale:                0.3669
Min. group size:     54        Log-Likelihood:       -470.5261
Max. group size:     72        Converged:            Yes
Mean group size:     60.8
------------------------------------------------------------
                   Coef.  Std.Err.    z    P>|z| [0.025 0.975]
------------------------------------------------------------
Intercept           1.364    0.260   5.240 0.000  0.854  1.874
gc[T.Parallel]     -0.002    0.067  -0.033 0.974 -0.134  0.130
gc[T.Serial]        0.002    0.067   0.028 0.978 -0.130  0.134
workload[T.Light]  -0.953    0.067 -14.153 0.000 -1.084 -0.821
workload[T.Medium] -0.762    0.067 -11.318 0.000 -0.894 -0.630
jdk[T.oracle]      -0.002    0.055  -0.030 0.976 -0.109  0.106
Group Var           0.506    0.455
============================================================
```

```
# GC × Workload interaction
model_interaction = mixedlm("log_power ~ gc * workload + jdk",
                            df,
                            groups=df["subject"])
result_int = model_interaction.fit()
print(result_int.summary())
```

```
                  Mixed Linear Model Regression Results
=====================================================================
Model:                  MixedLM       Dependent Variable:       log_pow
No. Observations:       486           Method:                   REML
No. Groups:             8             Scale:                    0.1304
Min. group size:        54            Log-Likelihood:           -226.51
```