Vrije Universiteit Amsterdam

Master Thesis

A Reproducible Energy Benchmarking Framework for Big Data Workloads

Author: Zhuoran Song (2803207)

1st supervisor:	Daniele Bonetta
daily supervisor:	Matthijs Jansen
2nd reader:	Tiziano De Matteis

A thesis submitted in fulfillment of the requirements for the VU Master of Science degree in Computer Science

July 18, 2025

Abstract

With the rising adoption of cloud-native analytics engines like Apache Spark, the energy implications of large-scale data analytics have drawn increasing attention (25). However, existing measurement practices often lack reproducibility, fine-grained visibility, and scalability (30), limiting our understanding of how workloads consume energy under different configurations.

This thesis presents the design and implementation of an automated framework for energy characterization of big data workloads. The framework integrates host-level power monitoring (via Scaphandre), virtualization management (via QEMU and Continuum), and a custom characterization pipeline to automate workload execution, metric collection, and power-phase analysis.

We validate the framework using SparkPi and apply it to the TPC-DS benchmark, performing a systematic characterization across five scaling scenarios. Our results reveal that query complexity has a stronger influence on instantaneous power draw than resource scaling and that increasing parallelism yields diminishing returns in energy efficiency beyond a certain threshold. These findings demonstrate the framework's ability to generate reproducible insights for energy-aware system design and workload planning.

Keywords: *Energy efficiency; Spark; Kubernetes; Benchmarking; Big data; Power measurement; TPC-DS; Scaphandre; Virtualization*

Contents

1

Introduction

In today's world, decisions are increasingly based on data. Digital infrastructure is no longer just a support system—it is now the core of modern economies and science (28). Like the railways of the industrial age or the electricity grids of the 20th century, today's cloud-native systems power critical operations across finance, healthcare, manufacturing, and research (6). This transformation is primarily driven by the rapid expansion of artificial intelligence (AI), big data analytics, and cloud computing (17). As more data is processed and stored, the demand for computing power rises sharply. According to the International Energy Agency (IEA), electricity use by data centers will more than double by 2030—reaching an estimated 945 terawatt-hours (TWh), up from 415 TWh in 2024. Most of this increase comes from the rapid use of AI, with the United States and China expected to account for nearly 80% of global data center energy consumption (17).

However, this wide-scale digital expansion brings a serious challenge: the energy cost of our digital habits. As people become more aware of environmental and energy constraints, the need to build systems that are not only fast or scalable—but also energy-efficient—has become more urgent (5).

Among these systems, big data processing platforms occupy a central role. These platforms are responsible for extracting insights from massive datasets and are widely used in industries ranging from finance to scientific research. As such, their energy consumption significantly contributes to the overall footprint of digital infrastructure (17).

In response, several sustainability programs have emerged. One prominent initiative is the Climate Neutral Data Centre Pact, a voluntary agreement supported by the European Commission (29). It commits members to targets such as improving energy efficiency, using carbon-free energy, saving water, and recycling equipment, with the ultimate goal of achieving carbon neutrality by 2030.

Despite this shift in awareness, most research in distributed systems still focuses on latency, system throughput, or security. The energy consumption of these systems is often overlooked (9). This is a surprising omission—systems that appear optimized in terms of performance may still consume excessive energy due to hidden inefficiencies in their architecture or automation.

Distributed data processing engines like Apache Spark have shown strong scalability and performance, making them popular for big data analytics. Apache Spark, in particular, is one of the most widely adopted big data frameworks in both industry and academia (30). However, their energy behaviour in real-world, container-based deployments remains poorly understood (25, 30). To address this knowledge gap, we must characterize how such systems consume energy under different deployment configurations—capturing how workload type, resource allocation, and runtime dynamics jointly influence energy usage (23). Identifying these patterns helps reveal hidden inefficiencies and trade-offs otherwise masked by conventional performance metrics.

Characterizing energy behaviour contributes to technical optimization and informs broader efforts to align digital infrastructure with sustainability goals. Given the growing reliance on data-intensive computing in sectors such as finance, healthcare, and scientific research, improving the energy efficiency of big data systems can support long-term resource stewardship without compromising computational capability (5).

This study presents a framework to address this gap by evaluating the energy behavior of big data workloads under different deployment configurations. We develop and validate a framework using Apache Spark as a representative system. Through this approach, the study aims to provide tools and insights that support the design of more energy-efficient data processing infrastructures.

1.1 Context

Although sustainability goals are increasingly emphasized at the policy level, energy visibility in real-world data processing systems remains limited. Platforms like Apache Spark are engineered for large-scale performance and throughput but offer limited built-in support for energy awareness (30). Existing monitoring tools often rely on coarse-grained metrics such as CPU utilization or execution time, which, while useful for performance diagnostics, do not fully capture the nuances of actual energy consumption (25, 30). Relying solely on these proxies can obscure hidden inefficiencies—systems may appear well-optimized in terms of throughput while still incurring unnecessary power costs.

The widespread use of containerization and virtualization technologies amplifies this challenge. These layers introduce an abstraction that complicates access to hardware-level telemetry and obscures the energy behavior of distributed workloads (2). As a result, system-level energy costs remain poorly characterized in practice.

To address this, emerging tools such as Scaphandre (14) attempt to bridge the observability gap. Scaphandre is an open-source exporter designed for integration with observability stacks like Prometheus and Grafana (7, 14). It leverages Intel's RAPL (Running Average Power Limit) interface—a hardware-level feature that estimates energy usage for CPU sockets and DRAM (13)—to collect power-related metrics at fine granularity. However, its application in real-world big data systems remains limited, and few studies have systematically evaluated its performance under diverse deployment scenarios.

1.2 Problem Statement

While the energy footprint of data-intensive systems is gaining attention, our understanding of how modern big data platforms behave in terms of power consumption—especially under containerized deployment—is still limited.

This thesis focuses on characterizing the energy use of Spark-based big data workloads deployed on Kubernetes.It aims to explore how deployment configurations—such as executor count, resource allocation (CPU and memory), and workload distribution—affect energy usage and how tools like Scaphandre—a RAPL-based energy monitoring exporter—can reliably capture power consumption.

Main Problem: There is a lack of a robust, modular, and reproducible framework for characterizing and evaluating the energy consumption of big data workloads, particularly those deployed using platforms like Apache Spark within containerized and virtualized environments.

While awareness of energy concerns is increasing, several practical and methodological challenges remain unresolved. First, energy monitoring tools are often difficult to integrate into modern big data systems and rarely offer the granularity needed to observe container-level behavior. Second, the absence of standardized benchmarks and configuration guidelines limits our ability to design meaningful energy-aware experiments. Finally, even when energy data is collected, it is seldom translated into actionable insights that can inform sustainable deployment strategies. These issues create friction at multiple points in the energy analysis workflow—from measurement to interpretation to application.

1.3 Research Questions

This thesis formulates the following research questions to address the main and sub-problems outlined above. These questions guide the conceptual design of the energy measurement framework, the selection and execution of representative experiments, and the interpretation of results to extract actionable insights.

- **RQ1:** How can a scalable and reproducible framework be designed to measure the energy consumption of Spark-based big data workloads running on Kubernetes?
- **RQ2:** What benchmark workloads and configuration parameters can be selected to meaningfully characterize the diversity of energy behaviors in such systems?
- **RQ3:** What practical insights can be derived from energy measurement experiments to support more sustainable design and deployment of data-intensive systems?

1.4 Research Methodology

This thesis follows a design-and-evaluation methodology composed of three stages. Each stage reflects a set of design choices guided by the goal of enabling reproducible and fine-grained energy analysis in containerized big data environments.

Framework Design: We designed a framework to capture the energy consumption of Spark-based big data workloads running on Kubernetes clusters within virtual machines. Apache Spark is chosen as the core data processing engine due to its widespread adoption in both academic and industrial big data pipelines (30). Kubernetes provides the orchestration layer, reflecting modern containerized deployment practices in production-grade environments.

To ensure accurate power monitoring, this study integrates Scaphandre (14), an RAPL-based, open-source energy monitoring tool that exposes low-level power metrics. Scaphandre was selected after surveying available monitoring solutions based on its ability to provide continuous, fine-grained energy data with minimal system overhead (12). A sharing mechanism is implemented to propagate host-level RAPL metrics into the virtualized environment, allowing energy measurement without disrupting container isolation. Scaphandre's metrics are scraped by Prometheus, which serves as the central time-series collector for structured energy data (22).

A preliminary validation is performed using the SparkPi example (3) to test automation stability and verify that the collected energy data reflects expected workload behavior. This step ensures measurement correctness before scaling to more complex benchmarks.

Benchmark Design and Execution: We use TPC-DS as the primary benchmarking suite due to its realistic and diverse SQL-based business analytics queries, making it well-suited for evaluating energy behavior under different system configurations. Details of benchmark selection rationale, query profiling, and configuration design are presented in Chapter 3.5.

Analysis and Insight Extraction: Energy-related metrics collected by Prometheus are exported as structured time-series data. These metrics are then parsed and aggregated using custom Python scripts to compute total energy consumption per workload run. Visualization is performed using Matplotlib (15), focusing on comparative plots that reveal energy-performance trade-offs across configurations. This offline analysis strategy supports reproducibility and allows flexible integration with other experimental data without relying on a live dashboard.

1.5 Thesis Contributions

This thesis delivers several concrete contributions to energy-aware big data computing by answering the above research questions. These contributions are relevant to researchers investigating energy-performance trade-offs and practitioners designing scalable and sustainable data systems.

- **Design of a Measurement Framework:** We design a reusable and automated framework for evaluating the energy consumption of big data workloads using Apache Spark, adaptable to containerized and virtualized Kubernetes environments (see **Chapter 3**). The framework provides a modular foundation for energy profiling in distributed data systems and can be extended to other frameworks beyond Spark.

- **Benchmark Selection and Evaluation:** We construct a curated set of benchmark-derived workloads—primarily selected TPC-DS queries—that reflect real-world analytics scenarios (see **Section 3.5**). These workloads are evaluated across diverse cluster configurations to uncover energy-performance relationships (see **Chapter 6**). The benchmark matrix captures a range of operational patterns, including strong and weak scaling, resource saturation, and query-level variation, enabling standardized cross-scenario comparisons of energy efficiency.

- **Execution and Practical Implications:** We conduct a comprehensive evaluation using the developed framework, running Spark workloads under varied deployment configurations (see **Chapter 6**). The results show that energy consumption is not always proportional to workload size or resource scale. For instance, over-fragmentation of executors or high shuffle loads in complex queries can result in significant energy overhead with limited performance benefits. These empirical insights underscore the importance of fine-tuning both system parameters and query plans when optimizing for sustainability.

1.6 Plagiarism Declaration

I hereby declare that this thesis is entirely my work, has not been copied from any other source (including individuals, online materials, or AI tools), and has not been submitted for evaluation elsewhere.

For details on VU Amsterdam's plagiarism policy, please refer to: https://vu.nl/en/about-vu/more-about/academic-integrity

1.7 Thesis Structure

This thesis is structured as follows:

1. **Introduction.** Introduces the research context, problem statement, research questions, methodology, contributions, and outlines the thesis structure.

2. **Background.** Reviews foundational concepts including reproducible deployment with Continuum, energy monitoring with Scaphandre, and an overview of benchmarking targets.

3. **Design of the Energy Benchmarking Framework.** Details the design requirements, system architecture, automated workflow, and the systematic construction of benchmark workloads and resource configurations.

4. **Implementation.** Describes the technical implementation of the framework, including VM setup, energy monitoring integration, benchmarking pipeline, and data processing methods.

5. **SparkPi-based Energy Measurement Validation.** Presents validation experiments using the SparkPi workload to test the framework's stability, accuracy, and responsiveness under controlled settings.

6. **Evaluation.** Provides a systematic evaluation of TPC-DS workloads under diverse scaling configurations, analyzing energy consumption patterns and deriving insights to answer the research questions.

7. **Related Work.** Reviews existing literature on energy monitoring methods, Spark workload characterization, and benchmarking frameworks, and highlights how this thesis addresses their limitations.

8. **Lessons Learned.** Reflects on practical challenges and key learnings encountered during framework development and experimentation.

9. **Conclusion.** Summarizes the thesis findings, explicitly answers the research questions, discusses limitations, and suggests directions for future work.

2

Background

This chapter introduces the technical context and key components that form the foundation of our energy measurement framework. We focus on three elements: (1) an automation toolkit for reproducible deployment, (2) the energy telemetry tool used for power measurement, and (3) the benchmarking workloads used to evaluate energy behavior in big data systems. The purpose of this chapter is to provide conceptual background rather than implementation specifics. Details of the experimental setup appear in later chapters.

2.1 Reproducible Deployment with Continuum

Continuum (18) is an open-source automation framework designed to support reproducible cloud-like environments for systems research. It allows users to define and provision multi-node virtual infrastructures through simple configuration files. The generated VMs are compatible with QEMU/KVM and can be automatically integrated with orchestration platforms such as Kubernetes.

For our purposes, Continuum provides a consistent foundation to deploy container-based big data environments in a virtualized setting. Its extensibility also allows for customization, such as integrating monitoring agents or mounting telemetry data across VMs, which is essential for reliable energy profiling.

2.2 Energy Monitoring with Scaphandre

Energy consumption in virtualized systems is difficult to observe directly due to abstraction layers between the physical host and containerized applications. To address this, we use Scaphandre (14), a telemetry agent that exposes power metrics based on Intel's Running Average Power Limit (RAPL) interface.

Scaphandre provides access to cumulative energy readings for CPU and DRAM domains and supports integration with observability stacks such as Prometheus. While real-time power metrics are theoretically available, they are often unreliable in guest operating systems and are used cautiously in this study. The focus remains on energy consumption as the primary metric of interest.

2.3 Benchmarking Targets: From Synthetic to Realistic Workloads

Two types of workloads are used to evaluate the measurement framework. The first is SparkPi (3), a deterministic CPU-intensive computation that estimates π using Monte Carlo methods. This workload serves as a lightweight validation tool to test measurement consistency and scaling sensitivity in a controlled setting.

The second and primary benchmark suite is TPC-DS (1), a widely used standard for evaluating the performance of decision support systems. TPC-DS includes a variety of SQL queries with diverse computational characteristics, reflecting real-world scenarios in business analytics. Its use enables a systematic study of how system configurations and workload patterns influence energy behavior in distributed processing environments.

3

Design of the Energy Benchmarking Framework

This chapter addresses **RQ1** — ***How can a scalable and reproducible framework be designed to measure the energy consumption of Spark-based big data workloads running on Kubernetes?*** by detailing the design of a modular and traceable benchmarking architecture. The primary focus of this thesis lies in measuring and analyzing energy and power consumption patterns of TPC-DS workloads under varied Spark-on-Kubernetes configurations.

So, we designed a complete system architecture to support these experiments. The framework integrates multiple subsystems—including Apache Spark, Kubernetes, QEMU-based virtual machines, and the Scaphandre energy monitoring tool—into a coherent pipeline for data generation, table creation, and query execution. The system operates within a controlled virtualized environment designed to ensure consistency, isolation, and reproducibility across experiments. Validation of the measurement pipeline is described in Chapter 5.

The remainder of this chapter is organized as follows:

Section 3.1 outlines the functional and non-functional requirements that guide the system design;

Section 3.2 presents a high-level overview of the system setup and virtualization environment;

Section 3.3 explains the architecture and monitoring pipeline, including how energy data is collected and aligned with execution phases;

Section 3.4 describes the automated workflow for running experiments, logging workload phases, and generating visual outputs.

Section 3.5 presents the experimental design, outlining the resource scaling strategies and query selection criteria used to systematically evaluate energy behavior across a range of representative big data workloads.

3.1 Design Requirements

To ensure the benchmarking framework fulfills its intended purpose—analyzing the energy and power behavior of Spark-on-Kubernetes workloads—we identify both functional and non-functional requirements that guide the system design.

3.1.1 Functional Requirements (FQ)

- **FQ1:** To understand how different deployment choices influence energy usage, the system must allow automated submission of Spark jobs under varied configurations—such as scale factors, instance counts, and resource settings.
- **FQ2:** Since this study aims to correlate execution behavior with energy patterns, the system must capture power and energy metrics at fine temporal granularity throughout each workload run.
- **FQ3:** Energy consumption cannot be properly interpreted without knowing what the system is doing at each moment. Therefore, the system must distinguish and timestamp execution phases such as data generation, metadata setup, and query execution.
- **FQ4:** To reduce post-processing overhead and ensure consistent analysis, the framework should generate power-over-time plots and per-phase energy charts automatically from raw data.

3.1.2 Non-Functional Requirements (NFQ)

- **NFQ1:** Experiments should be easy to deploy and repeat. To that end, the system must integrate with the Continuum framework, which handles VM provisioning and configuration in a reproducible way.
- **NFQ2:** Energy results are only meaningful if they can be reliably repeated. Therefore, the setup should produce stable output across trials with minimal sensitivity to noise or interference.

- **NFQ3:** For long-term maintainability and flexibility, the system architecture must be modular—each component should have a clearly defined responsibility and interface.

3.2 System Overview

The energy benchmarking framework is deployed within a virtualized environment hosted on a single physical server. Due to hardware resource constraints and the need for rapid experimentation, the system is intentionally kept minimal by deploying only two virtual machines. These QEMU-based virtual machines (VMs) are provisioned and configured using the Continuum automation framework. Since the goal of this study is to analyze energy and power consumption patterns—not to evaluate scalability or large-cluster performance—the simplified setup suffices for our purposes. To emulate a realistic container-based deployment while maintaining observability and control, we adopt a Spark-on-Kubernetes architecture within virtual machines rather than using a local or monolithic Spark setup.

Based on these experimental considerations, we designed the system architecture as shown in Figure 3.1, which benchmarks TPC-DS workloads under various Spark configurations. The architecture is organized into three primary layers:

- **Host Layer:** A physical machine runs the experiment environment and hosts all virtual machines. The energy monitoring tool Scaphandre is deployed at this level to collect system-wide power and energy data via Intel RAPL. Deploying Scaphandre directly on the host avoids virtualization-induced inaccuracies and provides consistent access to low-level power counters, addressing FQ2.

- **Virtualization Layer:** Two QEMU-based VMs are configured via the Continuum framework. One VM serves as the Kubernetes controller node, while the other is the worker node where Spark tasks are executed. This separation allows for container orchestration without interference from control plane activity, ensuring that all energy measurements reflect only the compute workload (FQ2, NFQ2).

- **Container Orchestration Layer:** Within the worker VM, Kubernetes schedules and launches Spark driver and executor pods. The driver pod manages task orchestration, while the executor pods carry out the actual computation. These pods execute TPC-DS SQL queries under different configurations using a fixed 10GB dataset that is generated and prepared only once before all experiments. Running Spark

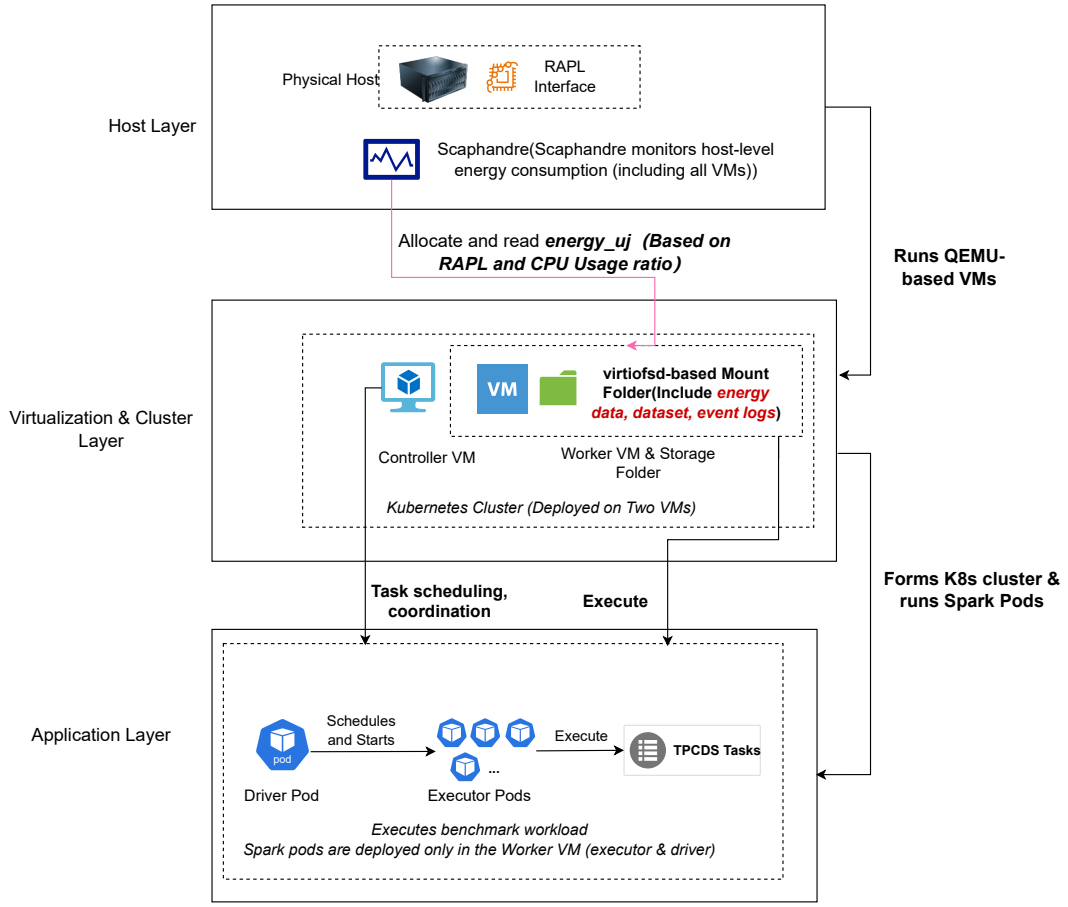


Figure 3.1: Overall architecture of the energy benchmarking framework for TPC-DS workloads on Spark-on-Kubernetes.

on Kubernetes reflects current best practices in cloud-native analytics systems and enables controlled manipulation of resource allocations at the pod level, satisfying FQ1 and FQ3.

This layered architecture ensures modularity and allows fine-grained observation of workload behaviors while enabling energy consumption measurement at the host level.

3.3 System Architecture and Monitoring Pipeline

To meet the needs for accurate energy monitoring (FQ2), phase-aware execution tracking (FQ3), and modular system design (NFQ3), we built a monitoring pipeline that is separated from the workload orchestration logic. This separation improves maintainability and

makes it easier to isolate potential measurement errors without interfering with workload execution.

At the core of our monitoring setup is Scaphandre(as shown in the Scaphandre component of Figure 3.1), a lightweight tool that reads power consumption metrics from Intel's Running Average Power Limit (RAPL) counters at the hardware level. These counters provide fine-grained energy data in microjoules for processor packages, cores, and DRAM. Scaphandre was selected for its minimal overhead, compatibility with virtualized environments, and ability to expose structured metrics suitable for time-series collection.

To associate energy data with each virtual machine (VM), we adopt a `virtiofsd`-based mount—QEMU's standard mechanism for sharing files between host and guest. During VM creation, the Continuum framework automatically sets up this mount, making each VM's energy trace accessible under `/var/lib/libvirt/scaphandre/` on the host(As shown in the Folder component of Figure 3.1). Inside this directory, each VM maintains an `energy_uj` file that updates in real-time as the VM runs. Our benchmark script samples this file periodically to build a continuous energy trace. This host-side mounting approach avoids intrusive instrumentation within the guest VMs while retaining accurate per-VM attribution.

To track execution behavior, we leverage Spark's built-in event logging system(As shown in the Folder component of Figure 3.1). During each query execution, event logs are collected to mark task boundaries and job progression. These logs are later used to segment the workload into sub-phases (e.g., initialization, shuffle, aggregation), which are then aligned with the corresponding energy trace. This structured logging method allows fine-grained temporal correlation without modifying application code or disrupting Spark's runtime behavior.

All energy and timing data are saved in raw form to ensure transparency and reproducibility. Post-processing scripts later align execution phases with energy readings, calculate total and per-phase energy consumption, and generate visual outputs such as power-over-time curves and energy bar charts. This analysis supports FQ4 by enabling consistent and automated interpretation of the measured data.

To meet the modularity goal (NFQ3), we split the system into three parts: (1) the workload scheduler, (2) the energy sampler, and (3) the visualization module. Each part has clear inputs and outputs and can be modified or reused without affecting the rest of the system.

The workload scheduler controls the automated execution of Spark jobs, including parameter configuration, resource assignment, and phase tracking. The energy sampler handles

periodic readings of the energy trace files exposed by Scaphandre and logs energy values throughout the experiment. The visualization module processes raw data and timestamps to generate figures such as power-over-time curves and segmented energy bar charts.

This modular design improves the framework's extensibility and maintainability, making it easier to debug, upgrade, or replace individual components when needed.

3.4 Automated Benchmark Workflow and Execution Control

We develop a benchmark workflow that runs seamlessly from start to finish with minimal manual intervention. This design enables efficient testing across a range of workload configurations while ensuring that each run is consistent and repeatable. It directly addresses requirements FQ1, FQ4, NFQ1, and NFQ2.

The workflow is driven by a set of predefined experiment configurations, each specifying parameters such as query identity, resource allocation, and instance count.[1] These configurations are automatically executed in sequence, allowing for controlled variation and batch experimentation.

Before launching a workload, the system performs an idle check to ensure that the host is in a thermally and computationally stable state. This precaution minimizes the influence of residual processes or temperature artifacts from previous runs. Each experiment focuses solely on the query execution stage, with input data and Hive metadata kept constant across all trials. A timestamp is recorded at the start of each run to enable alignment between execution phases and energy measurements.

During execution, Spark's event logging is enabled to collect detailed runtime metadata. These logs include time-resolved records of job and stage execution, which later support a fine-grained breakdown of query behavior.

After all experiments are complete, a post-processing module analyzes the collected logs and energy traces to construct energy profiles for each configuration. This analysis produces visualizations such as power-over-time curves and segmented energy bar charts. Results are systematically organized to facilitate comparison across experiments and parameter settings.As summarized in Algorithm 1, the entire workflow follows a structured and repeatable sequence—from idle checks and Spark job submission to energy monitoring and final analysis.

[1] All experiments share a common pre-generated dataset of 10GB scale. Data generation and metadata setup are performed only once, prior to the benchmark phase.

Algorithm 1: Execution Workflow for Benchmark Runs

Input: List of experiment configurations

Output: Organized energy profiles and visualizations

```
foreach configuration in the list do
    if Host is not idle then
        Wait until CPU and temperature stabilize;
    Launch Spark job with specified parameters;
    Record start timestamp for query phase;
    Enable Spark event logging;
    Monitor energy via Scaphandre during execution;
    if Experiment completes then
        Save energy trace and event logs;
Run post-processing to match logs with energy traces;
Generate visualizations (e.g., power curves, energy charts);
Organize results by configuration for comparison;
```

3.5 Benchmark Construction and Workload Characterization

This section contributes to addressing **RQ2** — ***What benchmark workloads and configuration parameters can be selected to meaningfully characterize the diversity of energy behaviors in such systems?*** by detailing the systematic design of benchmark workloads and resource configurations used in this study.

To systematically characterize the energy behavior of big data workloads, we base our experiments on the TPC-DS benchmark suite. TPC-DS is widely recognized as an industry-standard decision support benchmark, designed to model complex business analytics scenarios with realistic schema structures, diverse query types, and varying data access patterns (1). Compared to alternatives such as BigBench (8) or HiBench (16), TPC-DS offers a richer set of SQL-based analytical queries that closely reflect production workloads in modern data platforms, making it well-suited for evaluating both performance and energy implications.

We adopt the open-source tool `spark-data-generator` [1] to generate both the synthetic datasets and query templates in Spark-compatible format. However, the original implementation is tailored for large-scale clusters and is hardcoded to generate data starting

[1] `https://github.com/sacheendra/spark-data-generator`

from 100 GB. To make the tool suitable for our controlled, small-cluster setup, we modify the source code to support fine-grained scale factors (e.g., 1–10 GB) and enable configuration injection via external scripts. These adjustments allow us to align data generation with our resource constraints and experimental reproducibility goals.

In this study, a single 10GB dataset is generated and used throughout all experiments. Table creation and Hive metadata setup are also performed once before the benchmark phase. All subsequent experiments focus exclusively on the *query execution* stage using this pre-generated data.

3.5.1 Design of Resource Configuration Parameters

Our experiment matrix is structured to test how energy consumption responds to changes in resource allocation while keeping the dataset fixed. We design four types of configurations that reflect different real-world scaling patterns:

- **Type 1: Strong Scaling.** The problem size is fixed, while the number of executors or available cores is increased to test how energy changes with more resources.
- **Type 2: Resource-Constrained Load Scaling.** The resources remain constant, but the complexity of query execution varies across different TPC-DS queries.
- **Type 3: Weak Scaling.** The workload and resources grow proportionally, simulating a typical parallel scaling scenario.
- **Type 4: Distribution Scaling.** The total workload remains the same but is split across a varying number of executors, allowing analysis of energy trade-offs in parallel execution.

These configuration types were chosen to systematically cover scaling patterns commonly observed in big data clusters, ensuring the framework's ability to characterize both performance and energy implications under diverse deployment conditions.

3.5.2 Systematic Query Selection Strategy

To capture a broad spectrum of workload behaviors, we decided to select queries that not only exhibit distinct computational characteristics but also maintain practical relevance, ensuring coverage across common operation types such as I/O-intensive, compute-intensive, and memory-intensive workloads. We first performed preliminary dry-run profiling on all TPC-DS queries to evaluate their runtime, CPU utilization, memory consumption,

and I/O intensity. Based on this analysis, we systematically selected the following four representative queries from the TPC-DS suite:

- **Q3:** A lightweight and fast-running query with simple joins, used as a baseline to validate energy trace responsiveness and pipeline correctness.
- **Q5:** A scan-heavy query with multiple joins. It is I/O-intensive and stresses storage and network systems.
- **Q18:** A logic-heavy query involving subqueries and multi-table joins puts pressure on the query planner and CPU.
- **Q64:** An OLAP-style query with heavy sorting and aggregation, often bottlenecked by CPU and memory usage.

While this set does not exhaustively cover the entire TPC-DS workload space, it provides a practical and representative cross-section sufficient for energy characterization in this study.

4

Implementation

Building on the architectural design presented in Chapter 3, this chapter details the concrete implementation of the energy benchmarking framework for Spark-based big data workloads on Kubernetes.

The implementation aims to satisfy the following system requirements: accurate host-level energy monitoring (FQ2), phase-aware execution tracking (FQ3), and architectural modularity (NFQ3). Based on these goals, we have built a fully automated benchmarking pipeline that integrates workload submission, energy data collection, and visualization.

This chapter describes the implementation details of the following components:

- **VM Setup**: Extending the Continuum framework to automatically provision VMs with customised configurations, including memory backing and `virtiofsd`-based directory mounting.
- **Energy Monitoring**: Deploying Scaphandre on the host machine to collect RAPL-based energy metrics and using Prometheus-compatible exporters for data retrieval.
- **Benchmarking Pipeline**: Designing lightweight scripts to automate Spark job submission, collect timestamps for workload phases, and associate them with corresponding power data.
- **Data Post-processing and Visualisation**: Implementing mechanisms to process raw energy data and visualise power-over-time curves and cumulative energy bars across experiments.

4.1 VM Setup

We begin by preparing the infrastructure required for energy-aware benchmarking. The virtual environment is provisioned using the Continuum framework, which automates the deployment of QEMU-based virtual machines and their configurations. To support energy telemetry, specific adjustments are made to enable data sharing between host and guest.

Following the recommendations of Scaphandre, we enable `memoryBacking` and set up a shared file system via `virtiofsd`. These components allow telemetry output—such as energy usage data stored in `/var/lib/libvirt/scaphandre/` on the host—to be accessed by the virtual machines through mounted directories (e.g., `/var/scaphandre` inside the guest). This setup enables the collection of energy data externally while preserving VM isolation.

To ensure repeatability and consistent regeneration of virtual machines, the inventory system in Continuum has been updated to track VM states and include the necessary virtiofsd settings during domain configuration.

All infrastructure provisioning is automated through Continuum's domain generator, which has been extended to include these telemetry-specific options. These adjustments satisfy the non-functional requirement of modular deployability (NFR1), forming the foundation for the subsequent monitoring and benchmarking components.

4.2 Energy Monitoring

To collect energy-related metrics during workload execution, we deploy version 1.0.0 of the Scaphandre telemetry agent on the physical host machine. Scaphandre reads from Intel's Running Average Power Limit (RAPL) interface, which provides cumulative energy data for processor packages, cores, and DRAM. This setup enables hardware-level energy observation without being distorted by virtualisation artefacts.

In our implementation, we launch two parallel instances of Scaphandre: one using the `qemu` exporter and another using the `prometheus` exporter. The `qemu` exporter writes cumulative energy values for each VM to individual telemetry files under `/var/lib/libvirt/scaphandre/`. These files are accessed at the beginning and end of each experiment run, and the energy consumed during the workload is computed by taking the difference between the two readings. This simple delta calculation provides a reliable estimate of per-experiment energy consumption without requiring continuous sampling.

The `prometheus` exporter, in contrast, runs as a lightweight HTTP server on port 8081. It exposes host-wide power metrics—such as `scaph\_host\_power\_microwatts`—in a Prometheus-compatible format. Instead of running a full Prometheus service, we directly query this endpoint using simple HTTP requests (e.g., `curlhttp://localhost:8081/metrics`) to retrieve real-time power data.

Since Scaphandre does not support in-VM instantaneous power readings, we estimate the power usage attributable to each VM by combining two data sources: (1) the total host power obtained from the `prometheus` exporter, and (2) per-VM CPU usage reported by `/proc/stat`. This proportional attribution model enables the construction of power-over-time curves for each workload, even though measurements are collected only at the host level.

This monitoring setup fulfils the non-functional requirement of observability and reproducibility (NFR2), as it allows both cumulative and real-time energy data to be collected externally, processed reliably, and aligned with the execution phases of each experiment.

4.3 Benchmarking Pipeline

To enable consistent and reproducible experiments, we implement a lightweight benchmarking pipeline in Python that automates workload execution, energy data collection, and result processing. This pipeline coordinates the entire measurement cycle across multiple configurations, eliminating the need for manual intervention.

Each experiment begins by verifying system readiness. A pre-check ensures that the CPU usage on the host is sufficiently low (e.g., above 95% idle), reducing noise from previous jobs. Once idle conditions are met, the pipeline proceeds to launch a Spark workload via Kubernetes, using a configurable set of parameters: number of instances, core count, and memory per executor. All experiments are executed against a pre-generated 10GB TPC-DS dataset and existing Hive metadata, both of which remain fixed throughout the benchmark phase.

During each workload run, only the *query execution* stage is performed. Event logging is enabled in Spark to capture detailed execution traces, which include timestamps for jobs and stages. These logs are later parsed to extract fine-grained segments corresponding to operations such as *Scan*, *Join*, or *Aggregate*, allowing for detailed energy attribution.

Energy data is collected from two sources: cumulative readings from Scaphandre's `qemu` exporter and real-time power metrics from the `prometheus` exporter. The cumulative

energy is read at the beginning and end of each workload run, while real-time power data is pulled periodically by querying the exporter's HTTP endpoint.

To derive the power profile of each workload phase, we estimate per-VM instantaneous power based on the host's measured power and the VM's proportional CPU usage, obtained via `/proc/stat`. Specifically, we compute the VM power at time t using the formula:

$$P_{\text{vm}}(t) = P_{\text{host}}(t) \times \frac{C_{\text{vm}}(t)}{C_{\text{host}}(t)}$$

Where $P_{\text{host}}(t)$ is the power measured by Scaphandre, and $C_{\text{vm}}(t)$ and $C_{\text{host}}(t)$ represent the CPU time deltas for the VM and the entire host respectively. Power values are then interpolated and mapped to query job or stage windows defined by the event log timestamps. This results in a fine-grained power-over-time curve aligned with the workload timeline.

After each experiment is completed, energy and log data are stored with filenames encoding the experiment configuration. Output plots—such as stacked bar charts and line graphs—are generated for visual comparison. This automation enables large-scale experiments to be executed and analyzed in a systematic and reproducible manner.

4.4 Data Post-processing and Visualisation

The final stage of the benchmarking pipeline is dedicated to processing the raw energy and timestamp data collected during each experiment run. This post-processing step is implemented as a standalone Python module and is invoked after the workload is completed. It reads experiment-specific metadata—including workload parameters, phase timestamps, and cumulative energy logs—and transforms them into structured visual outputs.

The energy consumption during each workload phase is calculated by interpolating power values using the timestamp boundaries recorded during execution. The resulting data is then visualised in two primary formats: (1) power-over-time line charts that show the VM's estimated power profile across different execution stages and (2) stacked bar charts that compare cumulative energy consumption across experiments. All output plots are saved using configuration-encoded filenames and stored in a structured directory layout to facilitate batch comparisons.

While platforms such as Grafana are commonly used for real-time visualisation of Prometheus-exported metrics, they are not well-suited for reproducible, file-based batch analysis. Our experiments focus on offline benchmarking and comparative evaluation, which require complete control over visual styling, axis scaling, and data segmentation. To

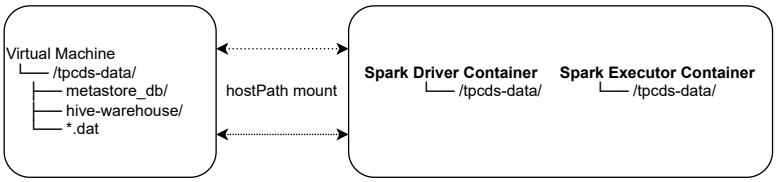


Figure 4.1: Shared `hostPath` mounting of `/tpcds-data` within a virtual machine, enabling consistent access to datasets, Hive metadata, and warehouse tables across Spark driver and executor containers.

meet these needs, we adopt `matplotlib` as our visualisation backend, allowing for precise formatting and a fully scripted generation of plots without any manual intervention.

This modular design ensures that visualisation remains decoupled from workload execution and can be independently reused or extended. As new benchmark configurations or workloads are introduced, the same post-processing logic can be directly applied, reinforcing the framework's reproducibility and scalability.

4.5 Data Volume Mounting and Hive Integration

To enable seamless access to pre-generated TPC-DS datasets during Spark execution, we implement a shared data volume mounting strategy within each virtual machine. Both the Spark driver and executor containers run inside the same VM, and a local directory on the VM—`/tpcds-data`—is mounted into each container using Kubernetes `hostPath` volumes. This ensures that input data, metadata, and output tables are accessible via a consistent file path across all workload phases.

This mounting setup supports all three benchmark stages—data generation, schema creation, and query execution—without requiring the duplication or copying of files between pods. The directory `/tpcds-data` contains the generated flat files, the embedded Hive metastore database, and the warehouse folder for structured tables. Figure 4.1 illustrates the shared directory structure within the virtual machine environment.

To integrate Hive functionality within Spark, we explicitly set the following Spark configurations:

- `spark.sql.catalogImplementation=hive` enables the use of Hive catalogue for table management;
- `spark.sql.warehouse.dir=/tpcds-data/hive-warehouse` specifies the directory for Hive-managed tables;

- `spark.hadoop.javax.jdo.option.ConnectionURL=jdbc:derby:;databaseName=/tpcds-data/metastore_db;create=true`
  initializes an embedded Derby metastore under the shared directory.

All these configurations are passed programmatically via `-conf` parameters during job submission. This approach eliminates the need for external storage solutions or distributed file systems, ensuring full automation of environment setup. By unifying data, schema, and catalogue into a single shared volume, the framework supports reproducible benchmarking across multiple runs and configurations.

5

SparkPi-based Energy Measurement Validation

This chapter presents the experimental validation of the proposed energy measurement framework. To verify its stability and responsiveness, we conduct a set of controlled experiments using SparkPi—a lightweight and deterministic workload—before applying the framework to more complex real-world scenarios. This section details the experimental environment and measurement workflow used in this validation phase.

5.1 Experimental Setup

The experiments were conducted on a physical server equipped with an Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, featuring two sockets, 20 physical cores, and 20 threads. Two virtual machines (VMs) were provisioned using the Continuum framework (18), which supports automated QEMU-based VM configuration. One VM was designated as the Kubernetes control plane, and the other as the worker node that exclusively executed all Spark jobs.

Each virtual machine (VM) was allocated 10 virtual CPUs and 32GB of memory, provisioned via QEMU/KVM full virtualization. The VMs use QEMU Virtual CPU version 2.5+, which emulates the host's physical processor (Intel Xeon Silver 4210 @ 2.20GHz) in a simplified virtual form. This setup enables the deployment of a fully isolated Spark cluster within a virtualized environment, allowing repeatable experimentation and accurate attribution of energy usage to individual VMs under controlled conditions (see Figure 5.1).

5.1.1 Software Stack

The software stack consisted of Apache Spark (Hadoop3, Version 3.4.4) deployed on a Kubernetes cluster configured with one control plane and worker node. Spark jobs were submitted in cluster mode using Kubernetes' native `k8s://` API endpoint. In cluster mode, the Spark driver runs inside the Kubernetes cluster—typically as a Pod scheduled alongside executors—rather than on the submission client. This approach ensures that all components of the Spark application, including driver and executors, are containerized and placed under Kubernetes resource control so that all parts of the Spark job stay inside the monitored environment, making it possible to capture the full energy usage of each job. Cluster mode also lets Kubernetes handle where to place the pods and how to allocate resources, which helps keep the experiments consistent and isolated.

Energy monitoring was performed using Scaphandre v1.0.0 (12), an open-source exporter designed to retrieve power consumption data from Intel's RAPL interface (13). Scaphandre was installed on the physical host and used in two modes simultaneously:

- The `qemu` exporter (11) exposed accumulated energy values (`energy_uj`) per VM by sharing telemetry files via a virtual filesystem;
- The `prometheus` exporter (22) streamed instantaneous power metrics (e.g., `scaph_host_power_microwatts` over HTTP, allowing continuous time-series collection.

Power data was queried periodically, and energy attribution to each VM was estimated based on its proportional CPU usage. This dual-exporter setup enabled cumulative and temporal energy tracking for Spark jobs inside the virtual cluster.

5.2 Measurement Pipeline

The measurement pipeline used throughout this thesis was originally developed for the SparkPi validation experiments, where it was designed to automate cluster setup, workload execution, energy monitoring, and result visualization. This early implementation served as a testing ground to verify the reliability and responsiveness of the energy measurement setup.

Building on this foundation, the same pipeline was extended and refined for TPC-DS benchmarks, as detailed in Section 4.3. The core mechanisms—such as idle-state detection, Spark job submission, and energy attribution based on proportional CPU usage—remain unchanged. Both qemu and prometheus exporters from Scaphandre are used to capture

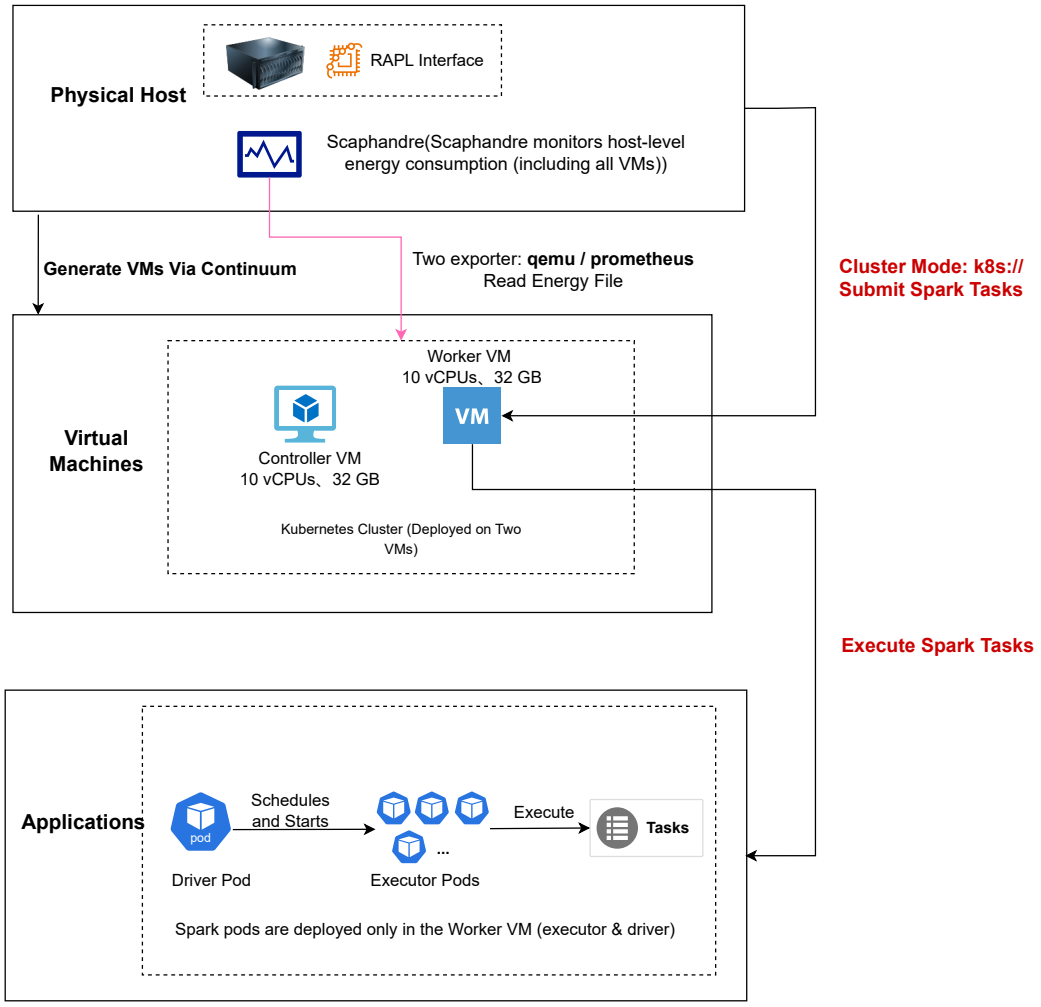


Figure 5.1: Experimental setup: Scaphandre collects VM energy via `qemu` and `prometheus` exporters, while Spark-on-Kubernetes runs inside a two-VM cluster.

host-level metrics, while execution segments are derived from Spark event logs and aligned with energy samples.

For SparkPi experiment, fewer semantic phases are logged, and the workload is lightweight and deterministic. The pipeline already supports all components of end-to-end automation, ensuring consistency across both validation and benchmarking phases.

5.3 Objective and Rationale

This experiment aims to verify whether our energy measurement pipeline can consistently and accurately capture power and energy variations across different computational patterns—specifically, changes in parallelism, load intensity, and executor distribution. We consider the pipeline reliable if it produces interpretable energy trends that align with known workload characteristics and scaling expectations.

To this end, we use SparkPi (3), a simple and deterministic workload that allows fine-grained control over input complexity while minimizing I/O interference. It enables controlled testing of computational stress across different scaling patterns.

We designed five representative scenarios: one idle baseline and four resource-driven strategies (strong scaling, increasing workload, weak scaling, and distribution scaling; see Section 3.5.1). These patterns simulate how systems behave under typical variations in core count, task complexity, and job distribution.

The goal is not to benchmark SparkPi's performance but to test whether our pipeline can consistently reflect the expected energy trends. Each scenario targets a distinct stress point, enabling us to validate whether power and energy metrics align with underlying execution logic.

5.4 Experiment Design & Results

Based on the scaling strategies introduced in Section 3.5.1, the SparkPi validation experiment was conducted under the configurations listed in Table 5.1. Each experiment is labeled by its scaling category and parameterization. The configurations vary the number of executor cores, allocated memory, workload size (via the number of point pairs), and Spark executor instances to simulate different stress levels on the system.

Four key parameters control each configuration:

- **Params**: Indicates the workload size, defined as the number of random point pairs generated by the SparkPi job. Larger values result in more computation and longer task durations.
- **Threads**: Refers to the number of CPU cores (executor cores) allocated to each Spark task. This simulates strong scaling behavior and affects parallel computation capability within each executor.

Table 5.1: SparkPi Experiment Configurations

Experiment	Params	Threads	Memory	Instances	Category Description
idle-baseline	–	–†	–†	–†	No workload submitted; VM remained idle with Spark environment preloaded. Used as energy baseline.
ss-1	100,000	1	1g	1	Strong scaling
ss-2	100,000	2	1g	1	
ss-4	100,000	4	1g	1	
iw-1	50,000	2	2g	1	Increasing workload
iw-2	100,000	2	2g	1	
iw-3	200,000	2	2g	1	
ws-1	50,000	1	1g	1	Weak scaling
ws-2	100,000	1	1g	2	
ws-3	200,000	1	1g	4	
sp-2in	50,000	1	1g	2	Split scaling (fixed total workload)
sp-4in	25,000	1	1g	4	

- **Memory**: Sets the per-executor memory allocation, which impacts JVM performance and garbage collection behavior.
- **Instances**: Specifies how many identical Spark executors are launched simultaneously. This enables testing weak scaling (increasing parallelism under constant per-task workload) and split scaling (fixed total workload split across multiple executors).

The experiment results—illustrated in Figures 5.1 to 5.3—demonstrate how energy usage and power profiles respond to different scaling patterns. Figure 5.2 presents the aggregated energy consumption across all SparkPi configurations. Figures 5.3 to 5.7 show the power-over-time curves for twelve representative configurations. These plots reflect how VM system power changes during job execution under different settings. Moreover, each plot's length of the x-axis also reveals each configuration's total runtime. The detailed parameter settings explanation for each configuration is listed in Table 5.1.

5.5 Takeaway

The SparkPi experiments offer strong evidence that our energy measurement pipeline can reliably capture power dynamics and energy consumption across varied scaling scenarios. Several findings support this validation:

- **Baseline stability**: The idle condition (`idle-baseline`) consumed only **1.4 J** (Figure 5.2), confirming that background power draw is minimal. The corresponding

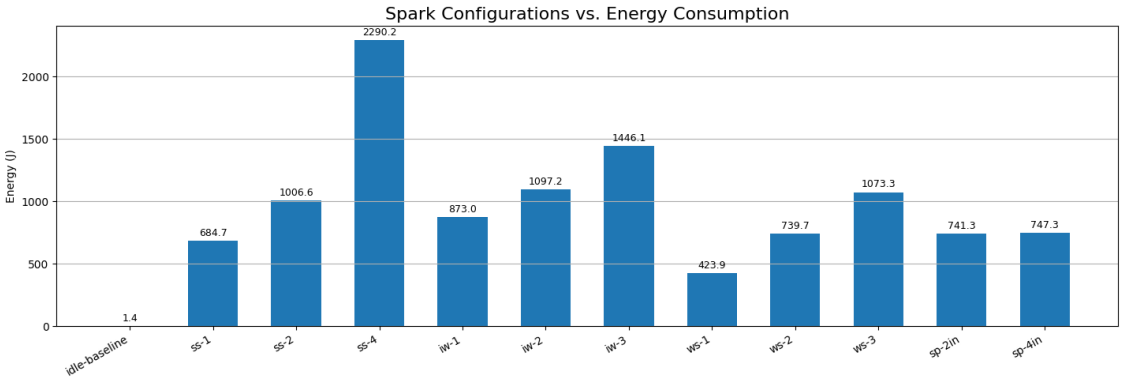


Figure 5.2: Comparison of sparkpi task energy consumption under different parameters

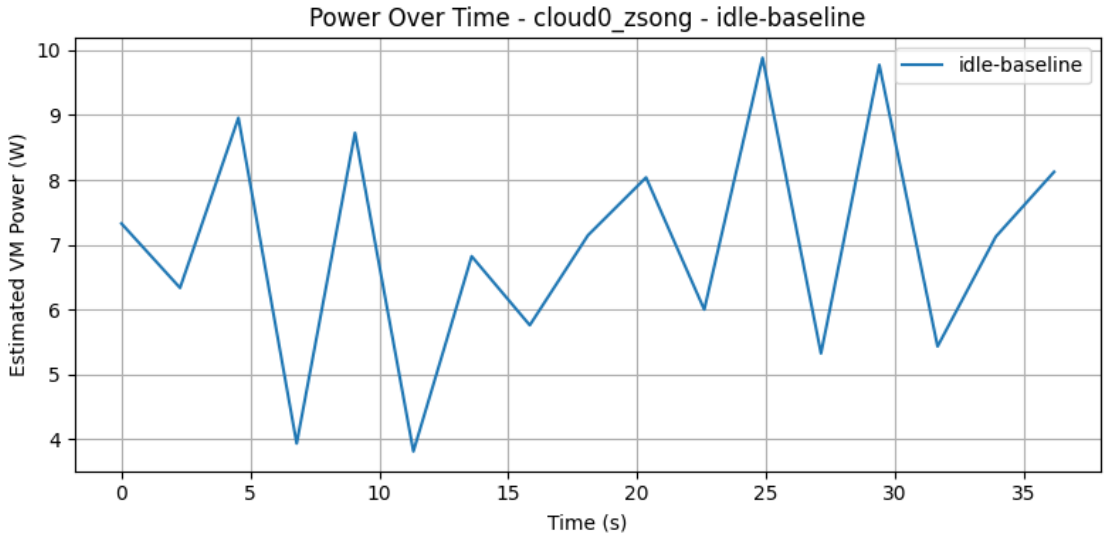


Figure 5.3: Power consumption of `cloud0_zsong` during the idle baseline phase.

power-over-time curve (Figure 5.3) remains flat between **6–9 W**, establishing a clean baseline for comparison.

- **Strong scaling validation (Figure 5.4)**: As executor cores increase from 1 to 4 (ss-1 → ss-4), power spikes become more prominent (peak up to **80 W**) and energy consumption grows from **684.7 J** to **2290.2 J**—an increase of over **3.3×** (Figure 5.2). Runtime drops significantly, but energy increases—confirming expected tradeoffs between power and time in strong scaling.

- **Workload scaling sensitivity (Figure 5.5)**: With fixed resources (2 cores), dou-

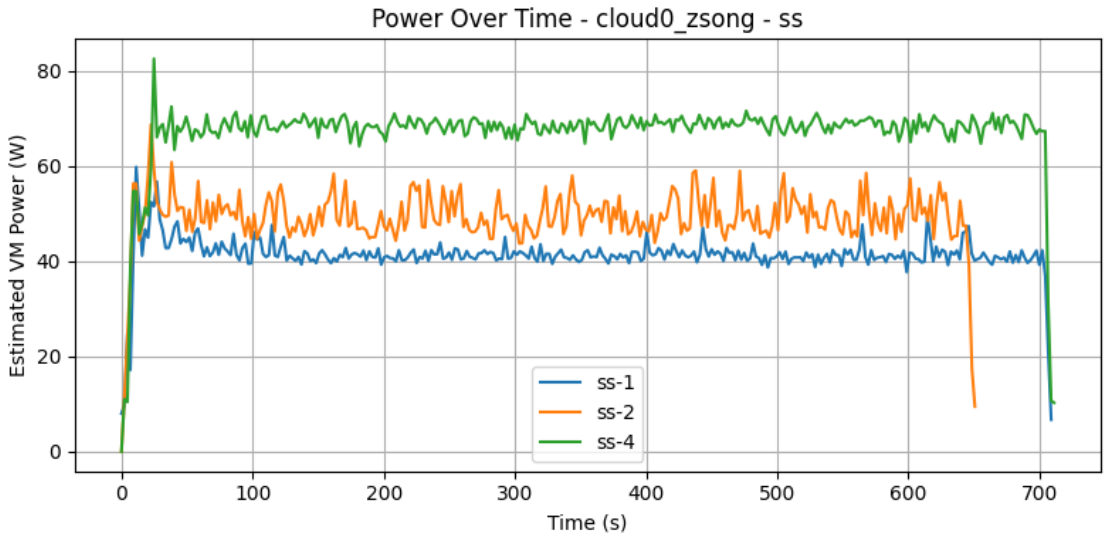


Figure 5.4: Power consumption of `cloud0_zsong` across strong-scaling experiments (`ss-1`, `ss-2`, `ss-4`).

bling and quadrupling the Workload (iw-1 → iw-3) results in energy rising from **873.0 J** to **1446.1 J**—a **1.7×** increase—while maintaining steady power profiles around **50–60 W**. This shows that the pipeline correctly tracks how heavier workloads increase energy even when power remains relatively stable.

- **Weak scaling reflection (Figure 5.6)**: Increasing executor instances under fixed per-task workload results in similar per-instance power (**45–55 W**) but increased cumulative load, with energy rising from **423.9 J** (ws-1) to **1073.3 J** (ws-3)—approximately a **2.5×** jump. Power curves exhibit stacked profiles as parallel instances increase, consistent with weak scaling behavior.

- **Executor distribution effect (Figure 5.7)**: Although the total Workload is fixed, splitting it across more executors (`sp-4in`) reduces runtime (under 200s vs. 400s) but increases peak power. Both configurations yield similar energy values around **741–747 J**, validating that the pipeline accurately attributes energy in distributed execution settings.

- **Power dynamics recognition**: All curves clearly capture *ramp-up → sustained load → shutdown* phases. The pipeline is sensitive to short bursts and differences in execution length—e.g., `sp-4in` completes in less than half the time of `iw-3`, which extends beyond **800s** (Figures 5.7 and 5.5).

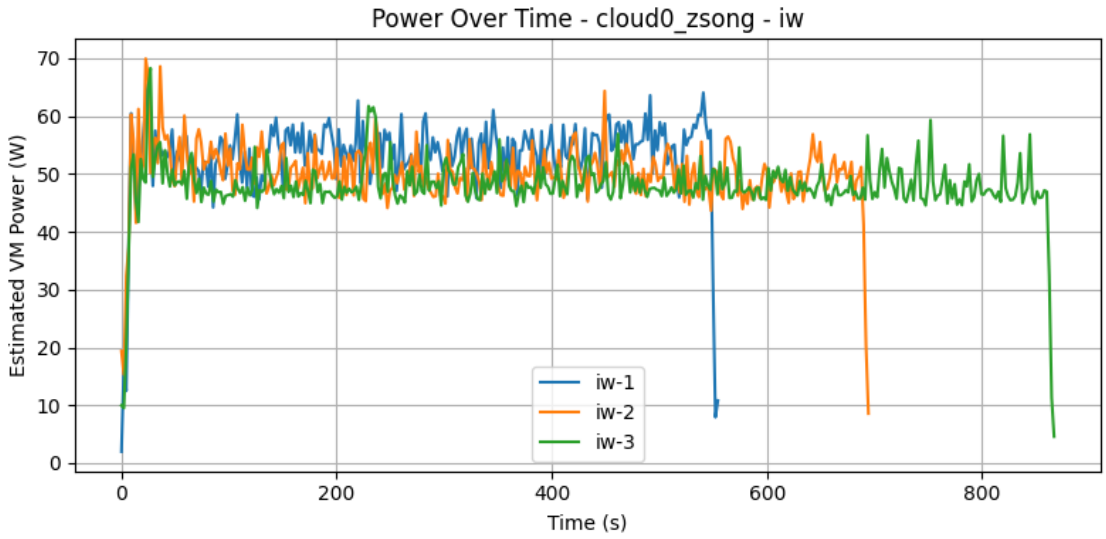


Figure 5.5: Power consumption of `cloud0_zsong` across increasing workload experiments (`iw-1`, `iw-2`, `iw-3`).

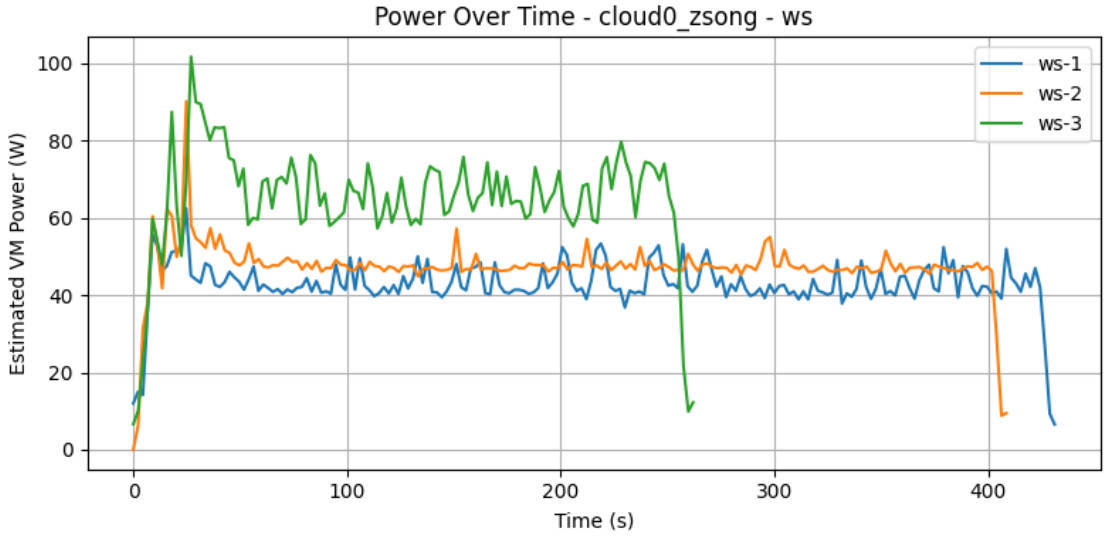


Figure 5.6: Power consumption of `cloud0_zsong` across weak-scaling experiments (`ws-1`, `ws-2`, `ws-3`).

Up to this point, the SparkPi results validate the pipeline's responsiveness to different stress patterns, its ability to preserve runtime variance, and its accuracy in quantifying both total energy and temporal power fluctuations. The insights serve as a confidence

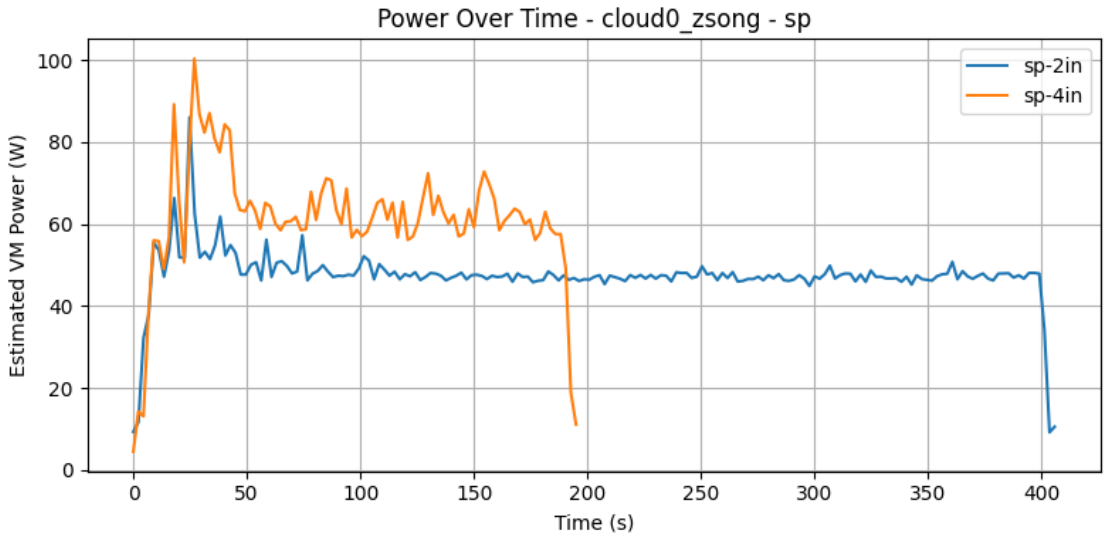


Figure 5.7: Power consumption of `cloud0_zsong` across split-scaling experiments (`sp-2in`, `sp-4in`).

foundation for applying the same pipeline to the more complex, multi-phase TPC-DS workloads in the next chapter.

6

Evaluation

This chapter evaluates the energy behavior of analytical workloads using the TPC-DS benchmark on Spark. Building on the validated measurement framework introduced earlier, we shift focus to more realistic and complex workloads representative of real-world data analytics pipelines. The goal is to understand how resource allocation strategies and query characteristics impact energy consumption across a variety of configurations.

In modern analytics infrastructure, engines like Apache Spark are widely used to power business intelligence, forecasting, and operational reporting (21). These platforms are designed to scale with more cores, more memory, and increased parallelism, promising faster results and higher throughput. But at what cost?

As organizations scale out their workloads, the energy required to answer even routine business questions—such as "What products sold best last week?" or "Which customers are likely to churn?"—can rise significantly. With cloud providers charging for both compute time and power usage and with sustainability becoming a growing priority, understanding these energy implications has become more than a theoretical concern.

In this chapter, we take a systems-level view of Spark-based analytics. Instead of focusing on abstract compute benchmarks, we turn to the TPC-DS workload—a representative suite of SQL queries used in real data platforms. Our goal is to understand how different ways of allocating resources, distributing tasks, or writing queries impact the system's overall energy footprint. As shown in Table 6.1, each query is run under controlled variations of resource scale, parallelism, and task distribution.

We ask: Is it always better to run jobs faster with more cores? What happens when multiple queries are batched? Can smarter query planning reduce power usage without sacrificing performance? These are the kinds of trade-offs that system designers and data

engineers face daily—and this chapter seeks to surface those dynamics through careful measurement and comparison.

Our analysis in this chapter targets the following research questions:

- **RQ2:** What benchmark workloads and configuration parameters can be selected to meaningfully characterize the diversity of energy behaviors in such systems?
- **RQ3:** What practical insights can be derived from energy measurement experiments to support more sustainable design and deployment of data-intensive systems?

To operationalize these questions, we examine five concrete sub-questions:

- Does increasing computing resources always lead to better energy efficiency?(Section 6.2)
- When system resources are fixed, how does increasing load affect power patterns?(Section 6.3)
- Can increased parallelism across multiple instances improve energy efficiency, or introduce new overheads?(Section 6.4)
- Is it more efficient to split the same workload across more workers?(Section 6.5)
- How does the structure and complexity of a query influence its energy footprint under the same system configuration?(Section 6.6)

To answer these questions, we analyze energy behavior across five configuration types. Each group corresponds to a specific scaling strategy and is evaluated using two perspectives: (1) total energy consumption (via bar charts with error bars) and (2) instantaneous power dynamics (via power-phase plots). The following sections first describe the experimental setup used to conduct these analyses, and then present key findings and insights from the evaluation of different scaling strategies and workload types.

6.1 Experimental Setup (for TPC-DS Experiments)

Building on the validated setup described in Chapter 5, the TPC-DS experiments inherit the same measurement environment while focusing on energy dynamics of analytical SQL workloads. This ensures continuity while shifting the focus to more complex analytical workloads.

The TPC-DS benchmark emphasizes energy behavior under varying resource constraints during SQL query execution. To isolate query-related energy consumption, the experimental design reuses a fixed dataset and metadata, and only the query phase is executed in each run. The setup is designed to observe how different Spark configurations affect energy use across various query types.

Static Data and Metadata Reuse

Before executing the benchmark, a one-time setup phase generates a 10 GB synthetic dataset using Spark's TPC-DS data generator and creates all required metadata tables. These are stored under the `/tpcds-data/` directory, which is mounted into the worker VM via shared hostPath. This setup ensures that subsequent query executions incur no I/O or compute overhead from table creation, allowing energy measurements to reflect the query phase exclusively.

Workload Configurations and Scaling Dimensions

Each experiment is defined by a tuple of parameters: the specific SQL query used (selected from TPC-DS official queries `q1` to `q99`), the number of Spark executors, cores per executor, memory per executor, and the number of parallel instances. These parameters are varied systematically according to the scaling categories introduced in Section 3.5.1.

Rather than repeating each configuration multiple times, the experiments are designed so that each setup has comparative value. Different configurations are grouped into logical families, enabling the contrastive analysis of energy usage patterns. The full list of experiment settings is presented in Table 6.1.

Query Execution and Phase-Specific Energy Attribution

To extract fine-grained energy metrics, only the *query execution* phase is performed in each experiment. Spark's event logging feature is enabled to capture detailed runtime traces, which are later parsed by a custom Python script to analyze execution behavior at the stage level.

This script processes Spark event logs in JSON format and extracts submission and completion timestamps for each stage. It then uses a semantic parsing function to identify the operation performed (e.g., `Scan`, `Join`, `Aggregate`, etc.) based on RDD scope metadata and stage names. A rule-based classification function assigns each stage to a semantic phase, with support for distinguishing operations such as `Exchange`, `Init(Map)`, or `TakeOrdered`.

Once all stages are classified, the script aggregates the duration of each semantic phase and retrieves total energy consumption from host-level Scaphandre logs. Energy is then

Table 6.1: Experiment Matrix for TPC-DS Query `q3-v2.4` and Variants

Type	Setting	Executors	Cores/Exec.	Memory/Exec.
Strong Scaling (Fixed Load, Increased Resources)				
T1-1	Minimal parallelism	1	1	4g
T1-2	Moderate scale-up	2	2	6g
T1-3	Single exec., higher cores	1	4	8g
T1-4	Balanced multi-executor config	2	3	6g
Resource Saturation (Fixed Resources, Increasing Repeats)				
T2	Constant 2x2 config with 6g mem	2	2	6g
Weak Scaling (Increased Resources with Problem Size)				
T3	1 to 4 executors, each 2 cores	1–4	2	4g
Distributed Execution (Same Load, More Executors)				
T4-1	All-in-one config	1	6	12g
T4-2	Split across 2 executors	2	3	6g
T4-3	Split across 3 executors	3	2	4g
Query Horizontal Comparison (Same Config, Varying Queries)				
T5	`q5`, `q18`, `q64`	2	2	6g

proportionally allocated across phases based on their timeshare. The final output includes a printed summary and a visualized bar chart of energy usage per phase.

This approach provides phase-specific attribution of total energy consumed per experiment, enabling detailed insight into how different operations contribute to the overall energy footprint of each query configuration.

6.2 Type 1: Strong Scaling — Does increasing computing resources always lead to better energy efficiency?

Experimental Design. In this setting, the workload remains fixed while computing resources are incrementally scaled up. The configurations include varying core counts and executor setups—from a single-core executor to multiple executors with more cores and memory. This simulates real-world questions around capacity planning: "Should I assign more resources to the same job to complete it faster and potentially save energy?"

6.2 Type 1: Strong Scaling — Does increasing computing resources always lead to better energy efficiency?

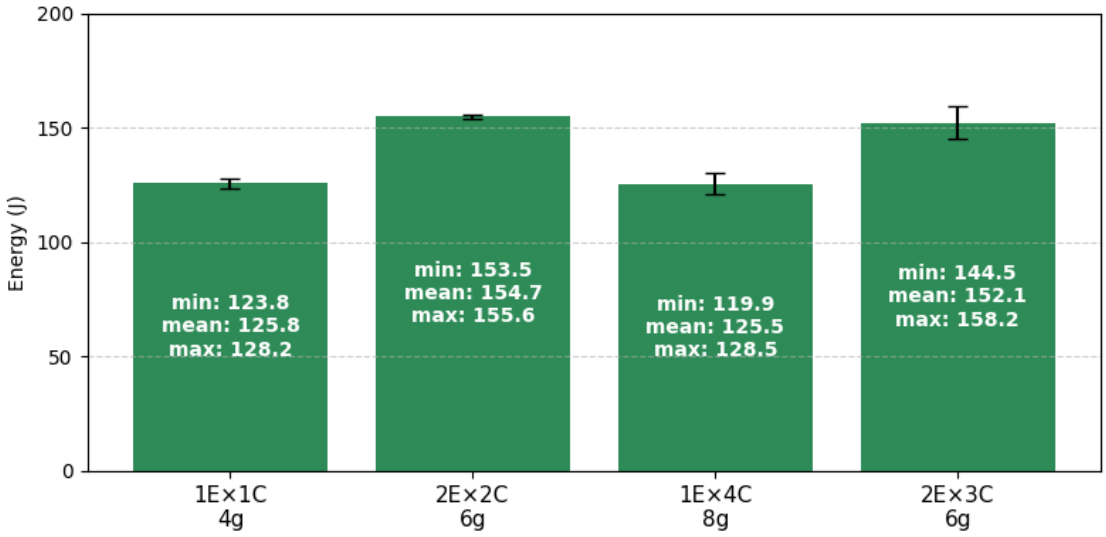


Figure 6.1: Energy Consumption with Error Bars for Strong Scaling Configurations (T1-1 to T1-4)

Rather than isolating individual parameters (e.g., only changing the core count or memory), the strong scaling setups reflect realistic deployment scenarios where multiple resource dimensions are typically adjusted together. For example, increasing executor count often comes with changes in memory allocation or task distribution in practice. While this design does not offer single-variable control, it enables the comparison of representative resource upgrade paths and captures practical system behavior under realistic scale-up strategies. This approach also aligns with how Spark is tuned in production—via co-tuning of cores, memory, and parallelism levels to balance efficiency and throughput.

Energy Comparison. Figure 6.1 presents the total energy consumption for the four strong scaling configurations. Surprisingly, energy usage does not consistently decrease with more computing resources. While T1-3 (1 exec × 4 cores) achieves the lowest energy (≈ 125.5 J), T1-2 (2 execs × 2 cores) results in the highest energy consumption (≈ 154.7 J). T1-1 (the most minimal setup) and T1-4 (balanced multi-executor) fall in the middle. This irregular pattern highlights the nonlinear trade-offs in strong scaling: more resources may reduce runtime, but also increase idle periods or coordination overheads.

Power Pattern Analysis. The time-series power plots reveal how different resource configurations affect power behavior during each query phase. Notably, larger configurations tend to complete phases more quickly but may exhibit higher peak power usage. Figures 6.4 show the power trends over time for T1-1 and T1-3 respectively. T1-1 exhibits a relatively lower power profile (mostly under 70W), but sustains this over a long duration

(≈67s), while T1-3 sees higher power peaks (up to 90W) but finishes much faster (≈45s). The net result is a lower total energy for T1-3, confirming that shorter execution time with higher instantaneous power can be more efficient.

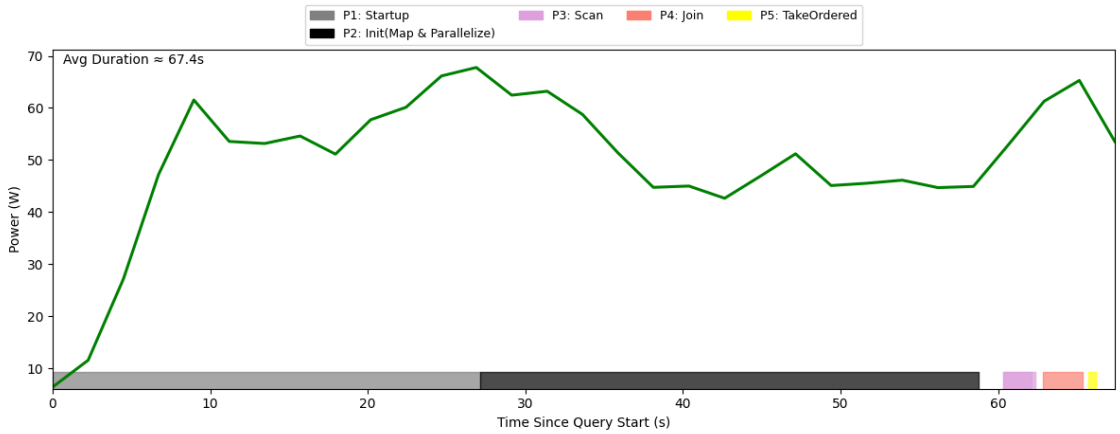
Startup Phase Interpretation. In all power plots, the initial segment labeled as "P1: Startup" corresponds to Spark's system-level initialization before any computational stages begin. This phase typically includes driver setup, executor registration, block manager initialization, and job planning, as inferred from Spark event logs. This conclusion is based on our analysis of Spark event logs, which record fine-grained timestamps and metadata for each system and job-level event before stage execution begins. Although no data processing occurs during this phase, it incurs measurable energy consumption and can last for 20–30 seconds depending on the experiment configs. All subsequent figures follow this convention for consistent interpretation.

Insights. These results have several practical implications: In real-world Spark deployments, increasing resource allocation may not always result in improved energy efficiency. Our findings show that T1-2, despite having more executors than T1-1, consumed the most energy—suggesting that distributed coordination overhead can offset the benefits of parallelism.

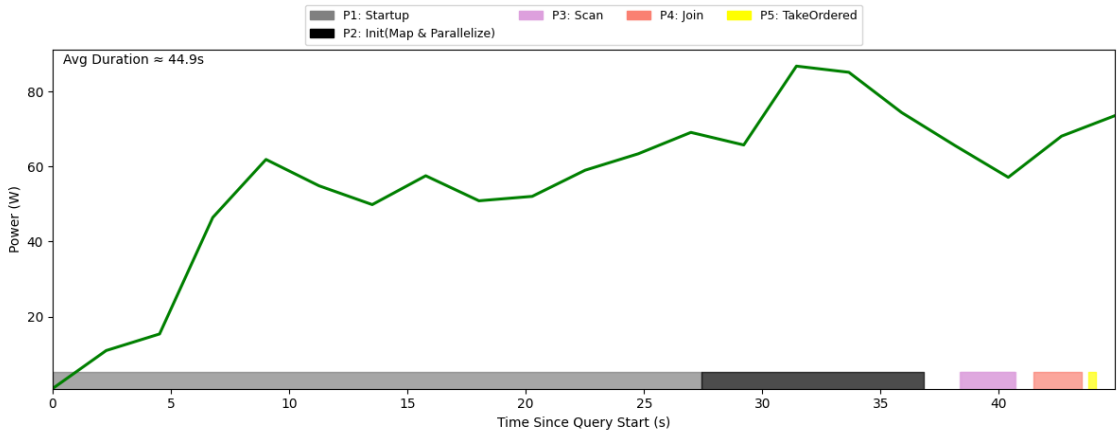
Moreover, T1-3 shows that a more concentrated configuration (fewer executors with more cores) can be both faster and more energy-efficient, even with higher peak power. This counters the intuitive assumption that high power is always wasteful—short bursts of intensive computing can be greener overall.

Such findings often carry important implications for cloud-based workloads, where autoscaling and serverless models typically charge based on both runtime and resource usage. They also illustrate the classic diminishing returns problem in strong scaling: once the problem size is fixed, adding more resources often leads to marginal runtime improvement at disproportionate energy cost.

These observations naturally lead to the next question: when system resources are fixed, how does increasing workload complexity affect energy patterns? To explore this, we turn to the second group of experiments (T2), where the query is executed repeatedly under identical resource constraints but with increasing load.



(a) T1-1 (1 exec × 1 core)



(b) T1-3 (1 exec × 4 cores)

Figure 6.4: Phase-wise power pattern comparison in Type 1: T1-1 vs. T1-3.

6.3 Type 2: Fixed Resources with Increasing Load — How does load intensity affect energy behavior when system capacity is constrained?

Experimental Design. In this group, the hardware configuration remains constant—2 executors, each with two cores and 6GB of memory—while the workload complexity increases by repeating the same query multiple times. This setup reflects common real-world limitations, especially in cost-conscious or shared environments where adding more resources is not always possible. It also mimics periodic workloads of different scales, such

6.3 Type 2: Fixed Resources with Increasing Load — How does load intensity affect energy behavior when system capacity is constrained?

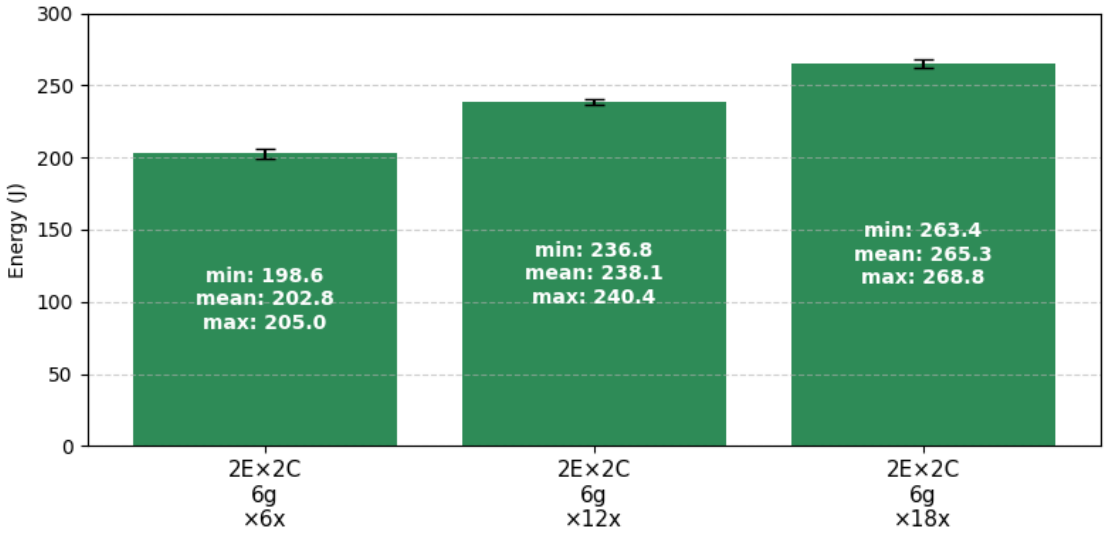


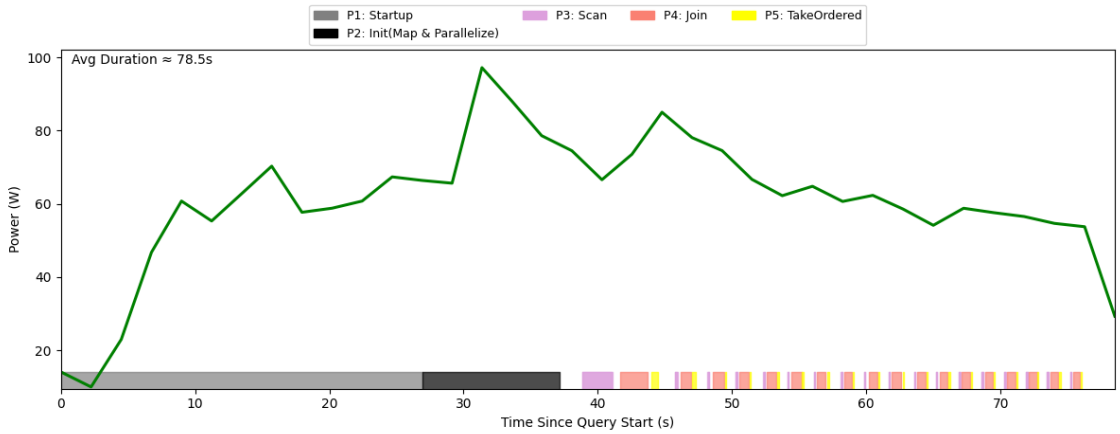
Figure 6.5: Energy Consumption with Error Bars (T2-1, T2-2, T2-3)

as daily, weekly, or monthly analytical jobs processed on a fixed infrastructure.

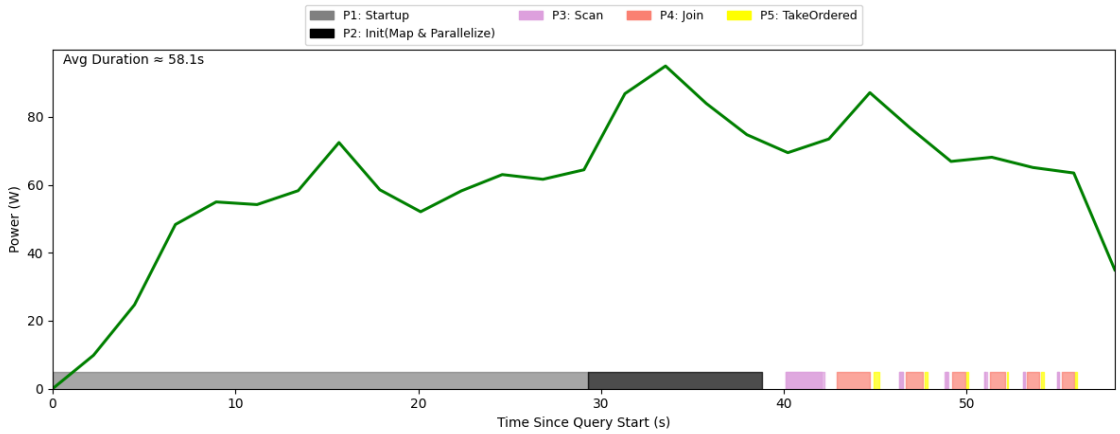
Energy Comparison. Figure 6.5 compares total energy consumption across increasing workload sizes. As expected, energy usage increases with heavier workloads: T2-1 (6 repeats) consumes ≈202.8 J, T2-2 (12 repeats) rises to ≈238.1 J, and T2-3 (18 repeats) peaks at ≈265.3 J. However, the growth is not strictly linear: the energy increase from T2-2 to T2-3 (≈27.2 J) is smaller than from T2-1 to T2-2 (≈35.3 J), suggesting a possible diminishing return effect.

Power Pattern Analysis. The phase-aligned power traces (e.g., Figure 6.8) show that increasing the number of query executions results in denser distributions of high-power stages. Interestingly, while the peak power remains relatively stable across T2-1 through T2-3 (≈95–100W), the duration of elevated power consumption extends noticeably. This implies that Spark, under fixed resources, does not aggressively scale instantaneous power per task but rather stretches the job timeline to accommodate heavier loads. This behavior may resemble a form of resource backpressure—i.e., the system continues to execute tasks at roughly the same intensity but queues or staggers their execution over time. Still, such interpretations should be treated with caution, as they rely on indirect inference rather than explicit profiling data.

Insights. These results point to several interesting patterns. As workload size increases under a fixed resource setting, total energy use naturally rises. However, the energy cost per unit of work decreases with scale, hinting at efficiency gains when grouping smaller



(a) T2-3 (large scale)



(b) T2-1 (small scale)

Figure 6.8: Phase-wise power trends under fixed-resource scaling: comparison between T2-3 and T2-1.

analytical tasks into batches. This batching effect may help distribute fixed costs—such as task scheduling and I/O initialization—more effectively.

Moreover, in cloud environments with burstable pricing, energy quotas, or thermal throttling, understanding how workload length impacts power duration—rather than peak power—can help developers better control total cost and system stability. Selecting batch sizes that align with the system's stable power envelope can lead to more predictable performance and energy profiles.

Another observation is the stable peak power across T2-1 to T2-3 despite increasing workload. This suggests that Spark maintains consistent execution intensity per core,

with larger workloads mainly extending the active runtime rather than increasing instantaneous demand. While this behavior offers predictability, it also opens up new avenues for optimization—such as smarter task scheduling, memory reuse, or adaptive planning—that deserve further exploration.

Building on these findings, the next experiment shifts focus: rather than increasing only the workload, we explore what happens when both the workload and available resources grow together.

6.4 Type 3: Weak Scaling — Does parallelization help maintain energy efficiency as workloads grow?

Experimental Design. This experiment group evaluates weak scaling behavior, where both workload size and computing resources increase proportionally. Specifically, the number of query repetitions is scaled with the number of executors—T3-1 runs three queries on one executor, T3-2 runs six queries on two executors, and T3-3 runs 12 queries on four executors. This design aims to simulate real-world settings where larger infrastructure is provisioned to handle growing job loads, such as monthly reporting on larger clusters compared to daily processing on smaller clusters.

Energy Comparison. Figure 6.9 summarizes total energy consumption under weak scaling. As expected, overall energy increases with workload size, but the growth is superlinear: T3-1 consumes ≈140.6 J, T3-2 jumps to ≈202.3 J, and T3-3 peaks at ≈321.1 J. Notably, the increase from T3-2 to T3-3 (≈+120 J) is significantly steeper than from T3-1 to T3-2, suggesting that beyond a certain point, parallel execution may introduce additional overheads that outweigh efficiency gains.

The phase-level energy breakdowns confirm this interpretation. In T3-1(Figure 6.12), the `Init(Map & Parallelize)` stage dominates energy use (≈88 J), with `Join` contributing a minor ≈21 J. In contrast, T3-3(Figure 6.12) shows a striking shift: while initialization energy remains high (≈94.5 J), the `Join` stage alone consumes ≈124.1 J—nearly six times higher than in T3-1. This phase-by-phase amplification reveals that energy inefficiency at scale is not uniformly distributed but instead concentrates on communication-intensive operations, such as distributed joins.

Furthermore, although the `Scan` and `TakeOrdered` stages also see absolute energy increases, their growth is far less dramatic, reinforcing the role of `Join` as the primary driver of overhead—and suggesting that scaling out resources without rethinking data shuffling patterns may worsen energy performance.

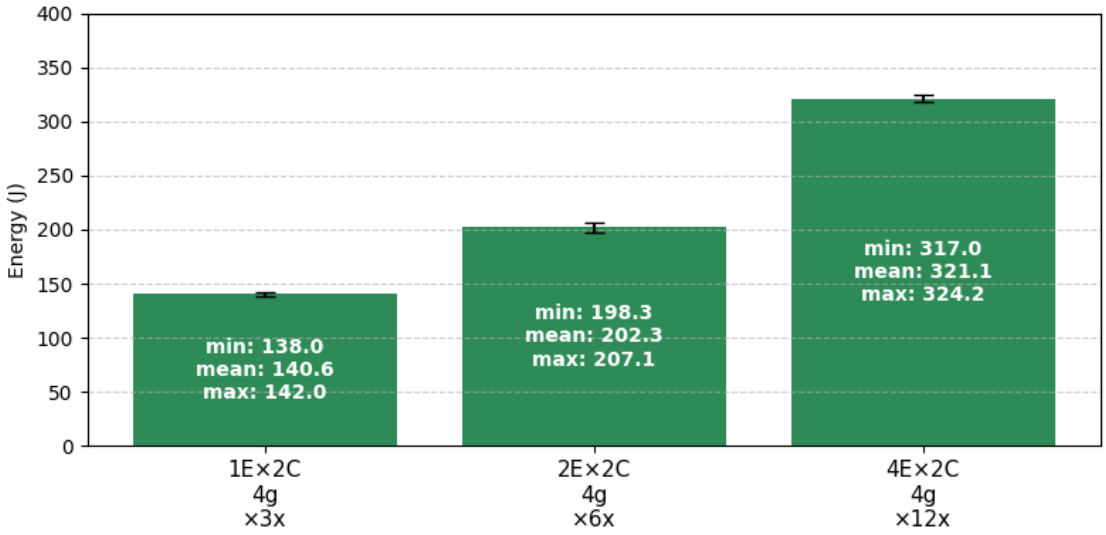


Figure 6.9: Energy Consumption with Error Bars under Weak Scaling (T3-1, T3-2, T3-3).

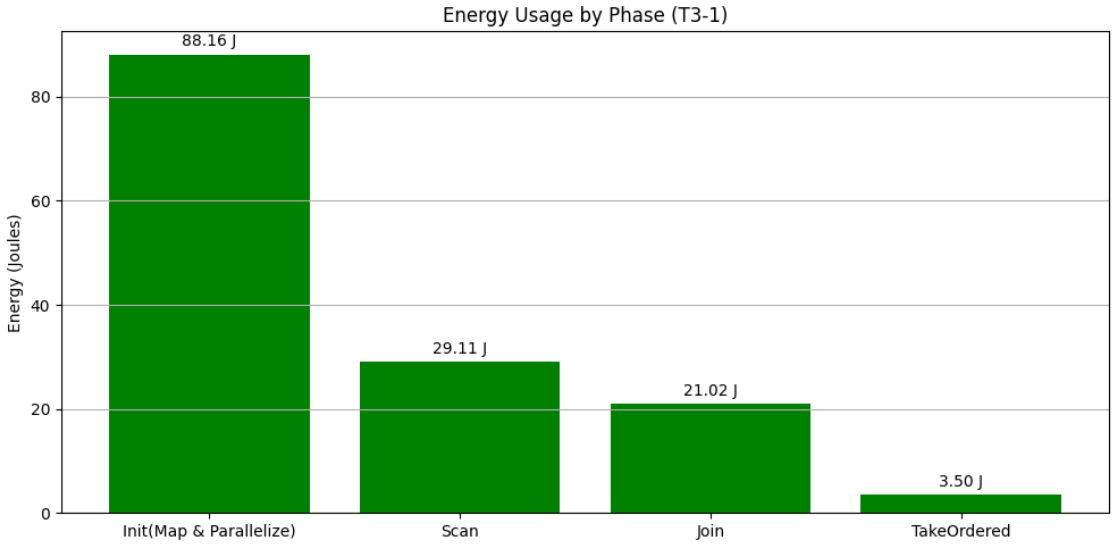
From this, we may tentatively conclude that weak scaling in distributed systems is especially sensitive to the nature of operations being scaled. For real-world deployments—particularly in large clusters running data-heavy SQL-like queries—optimizing join strategies (e.g., through broadcast joins, repartition tuning, or adaptive execution) may yield substantial and unexpected improvements in energy efficiency.

Power Pattern Analysis. Figures 6.15 illustrate the time-aligned power profiles for the smallest and largest configurations. Although peak power does increase (T3-3 surpasses 100W versus 80W in T3-1), it is the extended duration of elevated power that more significantly contributes to energy consumption. T3-3 takes ≈69.5 seconds to complete, compared to T3-1's ≈56.0 seconds, despite having four times the total core count. This indicates diminishing parallel efficiency, where per-query performance fails to scale proportionally with the addition of resources.

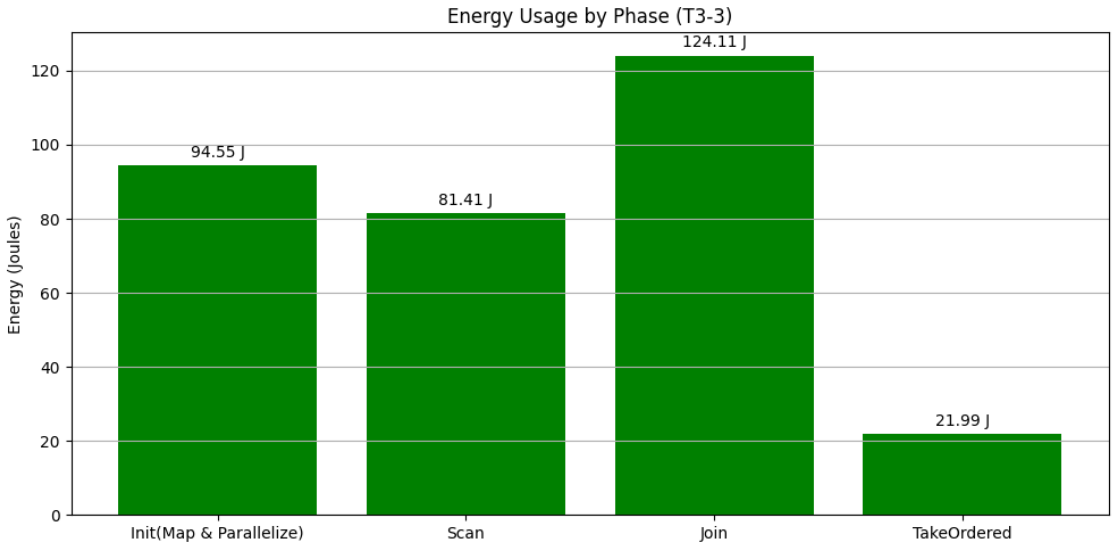
In addition, T3-3 exhibits noticeable power volatility, with sharp fluctuations aligned with frequent and densely packed execution phases such as repeated Join and TakeOrdered operations. The shaded overlays clearly show that power spikes correspond to shuffle-intensive and aggregation-heavy stages, which become more fragmented and frequent as the workload is split across more tasks. This confirms that the instability is not random but rooted in execution structure—specifically, synchronization points, stage barriers, and load imbalance across executors become more pronounced as parallelism increases.

Insights. These weak scaling results shed light on a familiar trade-off: pushing for

6.4 Type 3: Weak Scaling — Does parallelization help maintain energy efficiency as workloads grow?



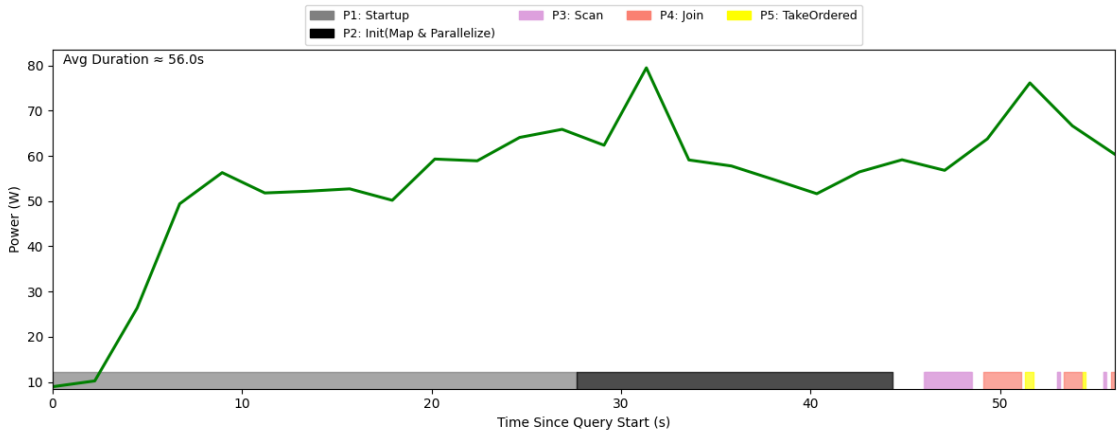
(a) T3-1



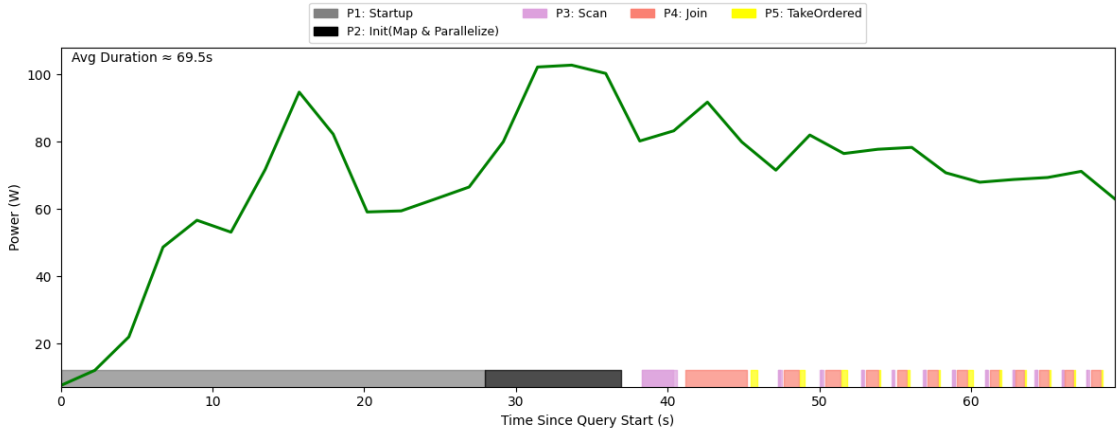
(b) T3-3

Figure 6.12: Energy distribution by phase under weak scaling: Join surge in T3-3 highlights scaling bottleneck.

higher throughput does not always go hand in hand with efficient energy use. Spark and similar frameworks are designed for scalable parallelism, but simply adding more workers does not guarantee improved performance or energy savings—especially when jobs involve extensive data shuffling or uneven task loads.



(a) T3-1 (small scale)



(b) T3-3 (large scale)

Figure 6.15: Phase-wise power trends under weak scaling: comparison between T3-3 and T3-1.

From a system design standpoint, the results serve as a reminder of the downside of over-provisioning. Take T3-3, for example—it uses the most resources but turns out to be the least energy-efficient per query run. This likely stems from the additional overhead required to coordinate tasks and manage communication between nodes. This pattern echoes real-world challenges in the industry, where scaling out too far often results in diminishing returns due to executor churn, network bottlenecks, or imbalanced workloads.

Interestingly, a comparison with Type 2 offers further nuance: while both types increased workload size, Type 2 maintained constant resource allocation and exhibited smoother power usage growth. Type 3, in contrast, added resources along with load and incurred

sharply rising energy costs—revealing the hidden inefficiencies of uncontrolled parallelism.

What if the workload remains constant, but we distribute it across a larger number of workers? We investigate this in Type 4.

6.5 Type 4: Fixed Total Load, Split Across Workers — Can distributing a fixed workload across more workers reduce energy usage?

Experimental Design. In practical scenarios—such as executing ad hoc analytical queries over a known dataset—engineers often face a resource allocation dilemma: should available computing power be consolidated into a few large executors or distributed across smaller ones? This experiment explores that question. All configurations process the same workload exactly once but with varying degrees of parallelism: T4-1 uses one executor with six cores, T4-2 splits this into two executors with three cores each, and T4-3 uses three executors with two cores each.

Energy Comparison. As shown in Figure 6.16, total energy consumption rises as the workload is spread across more executors: T4-1 consumes approximately 126.5 J, T4-2 increases to 154.8 J, and T4-3 reaches 181.1 J. The most significant increase occurs between T4-1 and T4-2, indicating that the introduction of multi-executor coordination incurs a steep initial energy overhead. The growth continues from T4-2 to T4-3, albeit with a smaller marginal cost, suggesting a diminishing energy efficiency as parallelism scales up. This pattern reflects the rising cost of task scheduling, shuffle communication, and coordination that can outweigh the benefit of reduced load per executor.

Power Pattern Analysis. Figures 6.19 further illustrates this overhead. While the execution time remains nearly the same across configurations (≈42–45s), T4-3 exhibits a noticeably higher and more sustained power profile—peaking above 100W compared to 90W in T4-1. This indicates that increasing the number of executors elevates baseline power usage without reducing runtime.

Additionally, the power curve of T4-3 is more irregular, with sharper fluctuations between stages. This suggests the need for more frequent inter-executor communication and finer-grained task transitions. That is, the workload is split into smaller units, but the added synchronization effort prevents any significant performance gain—resulting in higher energy expenditure.

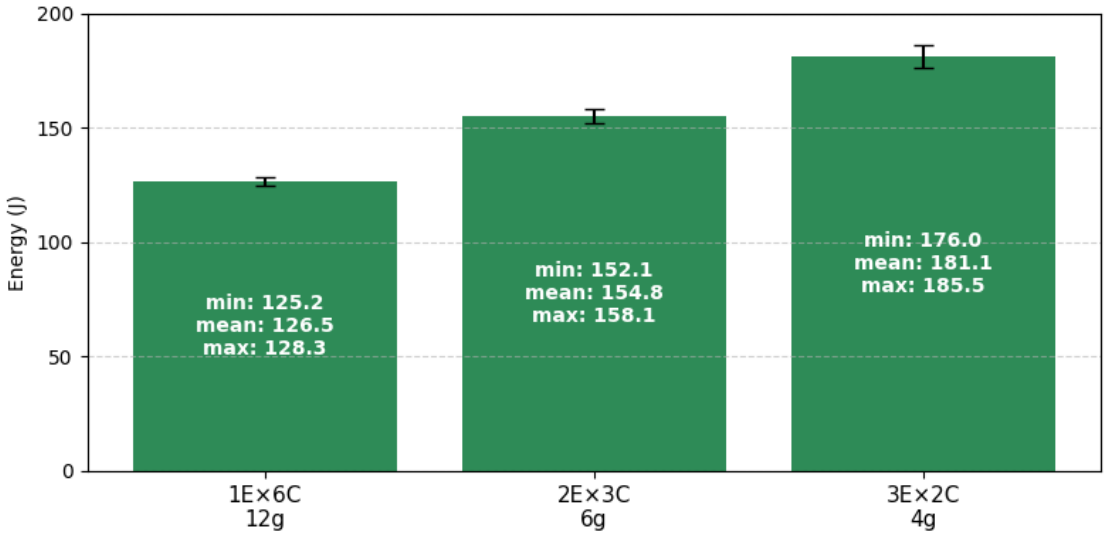


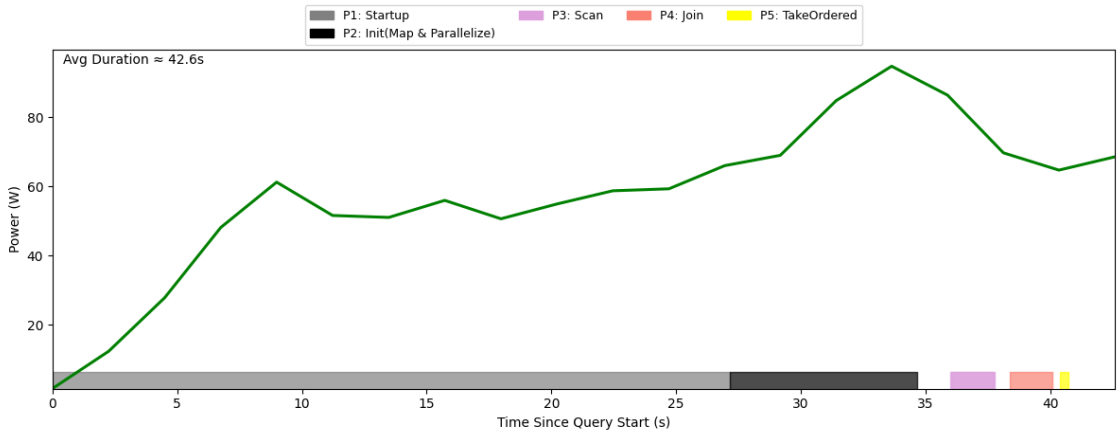
Figure 6.16: Energy consumption across T4 configurations.

Insights. When the total workload is fixed, increasing the number of executors can degrade energy efficiency. Our results show that additional parallelism may lead to unnecessary coordination overhead, especially when the job is not CPU-bound. For users running medium-scale Spark workloads, consolidating tasks onto fewer, more powerful executors can reduce communication costs and lower total energy usage. This insight is useful for systems that need to operate within tight energy, thermal, or budget limits.

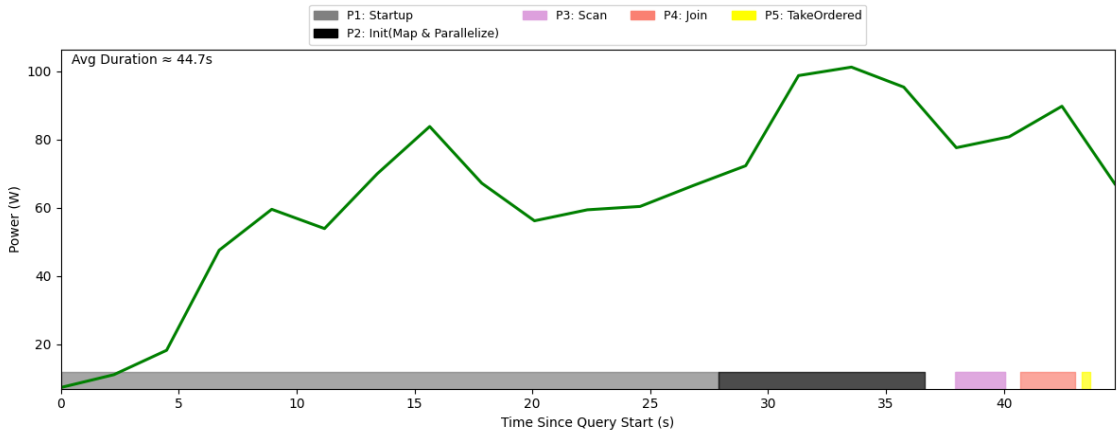
6.6 Workload-Specific Energy Insights

From Section 6.2 to Section 6.5, we examined how energy consumption varies with different resource configurations using a fixed query—`q3`. This query serves as our **baseline workload**, as it integrates a balanced mix of data initialization, scan, and join operations and avoids extreme computational skew. As such, it provides a stable reference point for evaluating resource-based scaling strategies without introducing variability from the query logic itself.

However, production-scale analytics seldom rely on a single representative query. In real-world use cases—such as TPC-DS benchmark deployments, dashboard aggregations, or report generation pipelines—workloads can vary dramatically in structure. To better understand this variation, we expand the investigation to compare queries with diverse logical structures but executed under identical cluster settings.



(a) T4-1 (single executor with 6 cores)



(b) T4-3 (three executors with 2 cores each)

Figure 6.19: Phase-wise power trends in fixed-load split execution: T4-1 vs. T4-3.

We selected `q5`, `q18`, and `q64` from the TPC-DS benchmark to reflect three common workload types encountered in practice. These queries represent contrasting execution plans: `q5` emphasizes joins and group-by aggregations, `q18` is dominated by table scans, and `q64` contains both wide scans and deep aggregations. This selection enables a structured comparison across common SQL workload archetypes seen in enterprise and cloud-based analytics. Our goal is to assess how **different workload structures manifest in energy usage** and which operations contribute most to power draw and inefficiency.

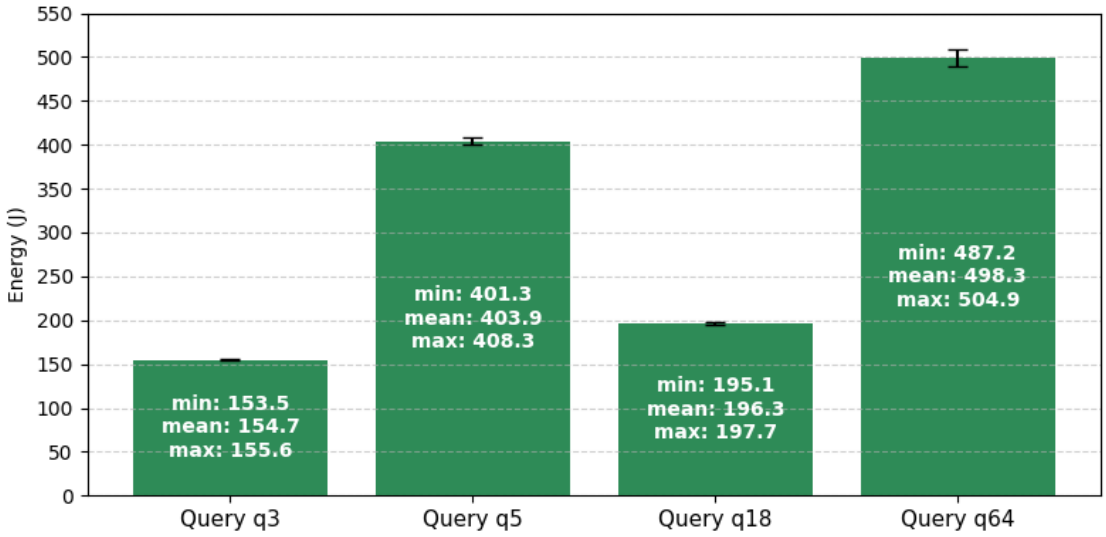


Figure 6.20: Total energy consumption across workloads: T1-2 (q3), T5-q5, T5-q18, and T5-q64.

6.6.1 Total Energy Consumption and Phase Breakdown

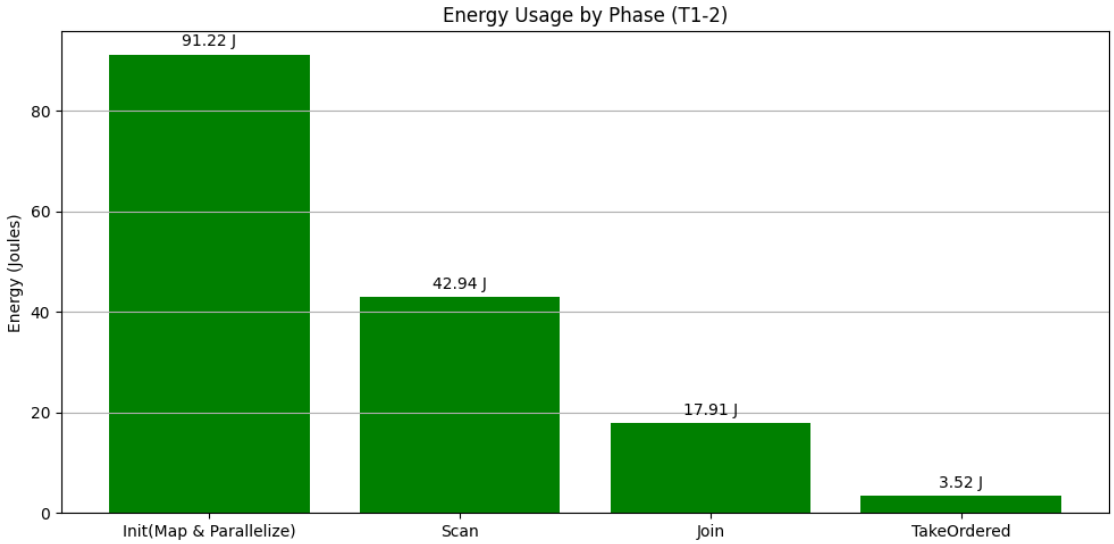
Figure 6.20 shows substantial disparities in total energy use despite identical hardware:

- **T1-2 (q3)**: ≈154.7 J
- **T5-q18**: ≈196.3 J
- **T5-q5**: ≈403.9 J
- **T5-q64**: ≈498.3 J

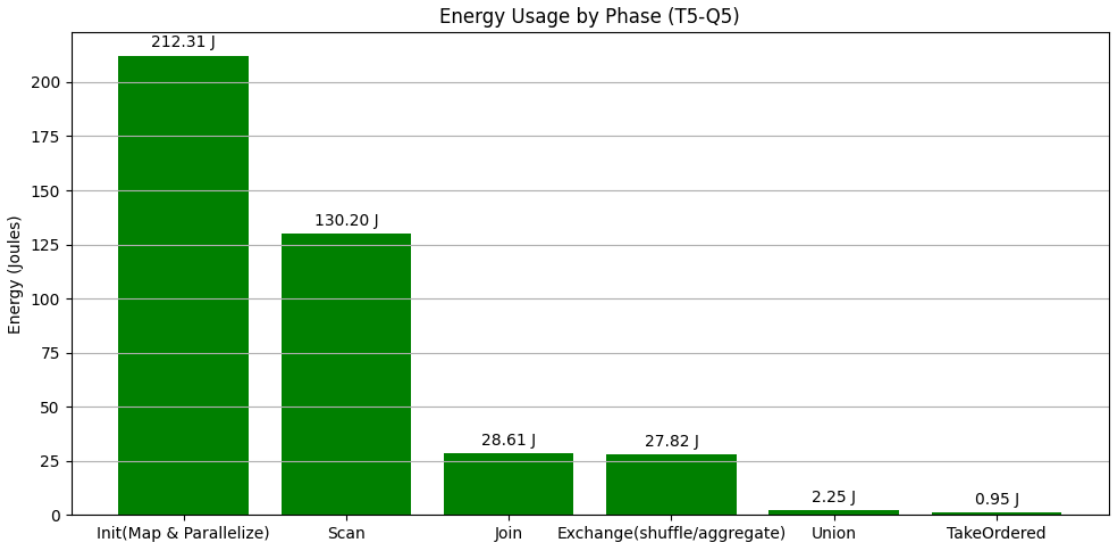
We can clearly see that the energy consumption differs by more than 3× between the lightest and heaviest queries, suggesting that the choice of workload structure itself is a major factor influencing overall energy use.

Phase-wise breakdowns (Figures 6.23 and Figures 6.26) reveal the source of these gaps:

- **q3 (T1-2)**: Energy is concentrated in `Init(Map & Parallelize)` and `Scan`, with a modest `Join`.
- **q18 (T5-q18)**: Almost exclusively scan-driven, with ≈130 J on `Scan`, and no `Join` or `Exchange` overhead—showing that queries with minimal coordination remain energy-light.



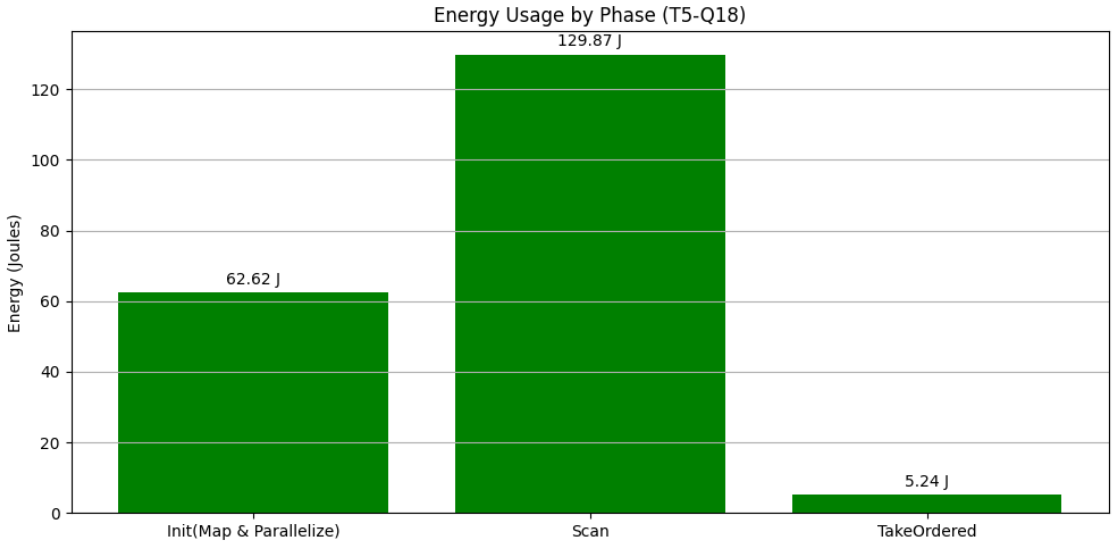
(a) T1-2 (q3 baseline)



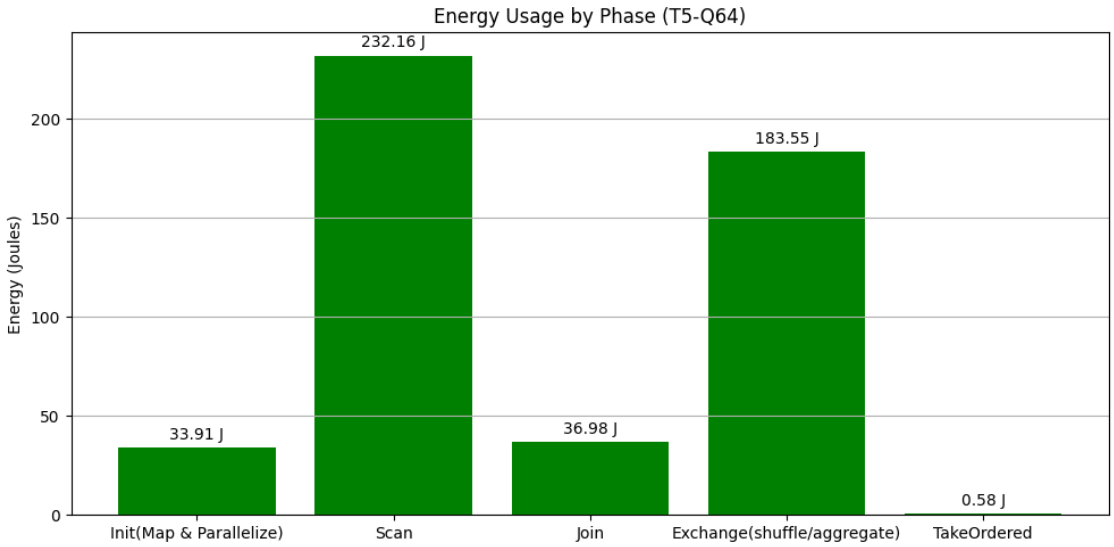
(b) T5-q5

Figure 6.23: Phase-wise energy usage: T1-2 (q3 baseline) vs. T5-q5.

- **q5 (T5-q5)**: Combines ≈130 J on `Scan`, ≈29 J on `Join`, and ≈28 J on `Exchange`, representing the energy cost of combining joins with shuffle-heavy grouping logic.
- **q64 (T5-q64)**: Has a dual energy peak: ≈232 J on `Scan` and ≈184 J on `Exchange`, with moderate join cost. This suggests that wide shuffles and large intermediate



(a) T5-q18

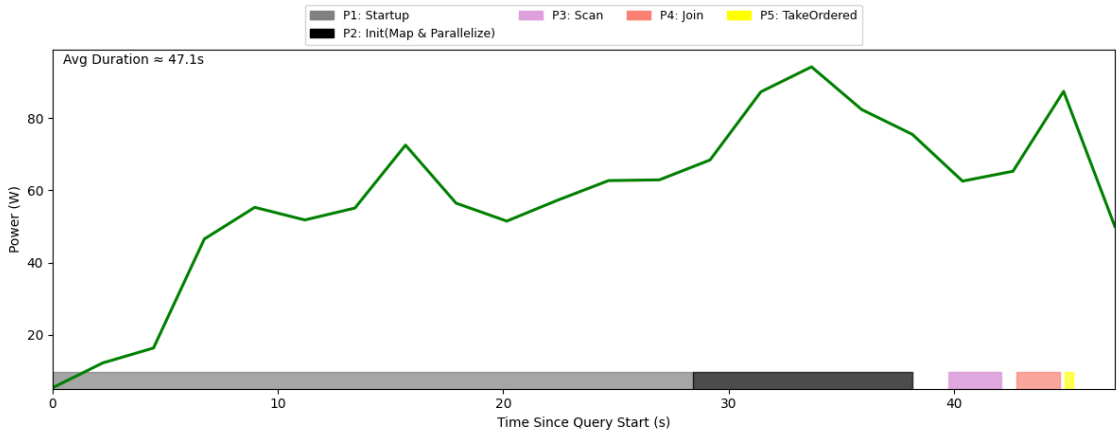


(b) T5-q64

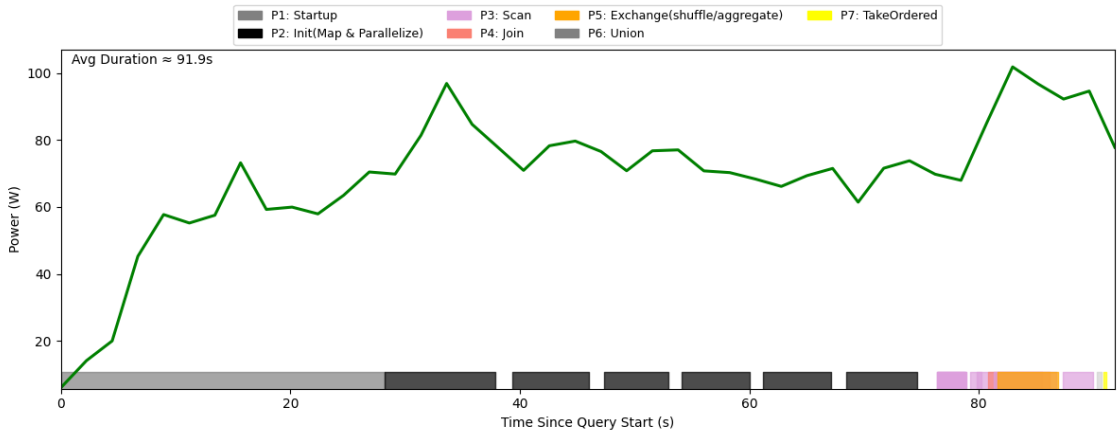
Figure 6.26: Phase-wise energy usage: T5-q18 vs. T5-q64.

result sets are key contributors to high-energy draw.

From these results, we can preliminarily find that it is not the number of joins alone that drives energy usage, but the cost of shuffling and materializing large aggregates across executors. In other words, coordination and intermediate data movement dominate energy



(a) T1-2 (q3 baseline)



(b) T5-q5

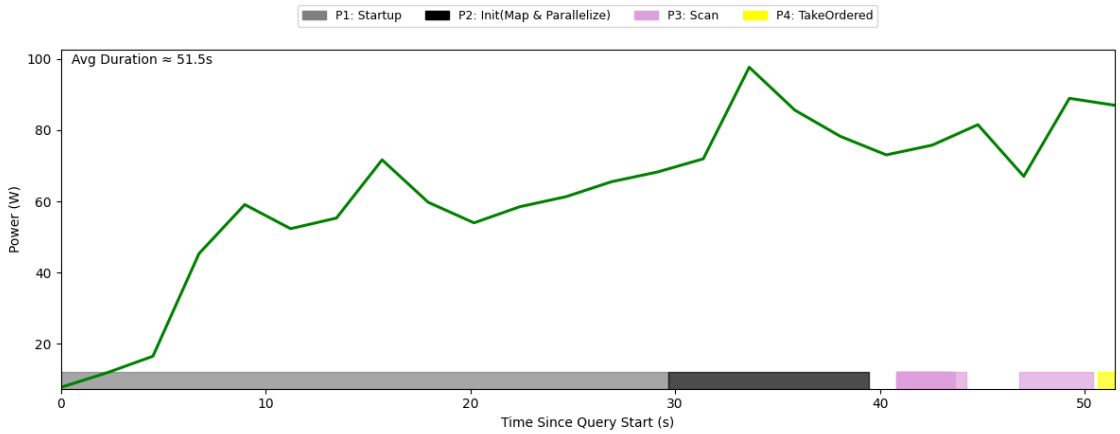
Figure 6.29: Power over time with phase highlighting: T1-2 (q3 baseline) vs. T5-q5.

overhead in complex queries.

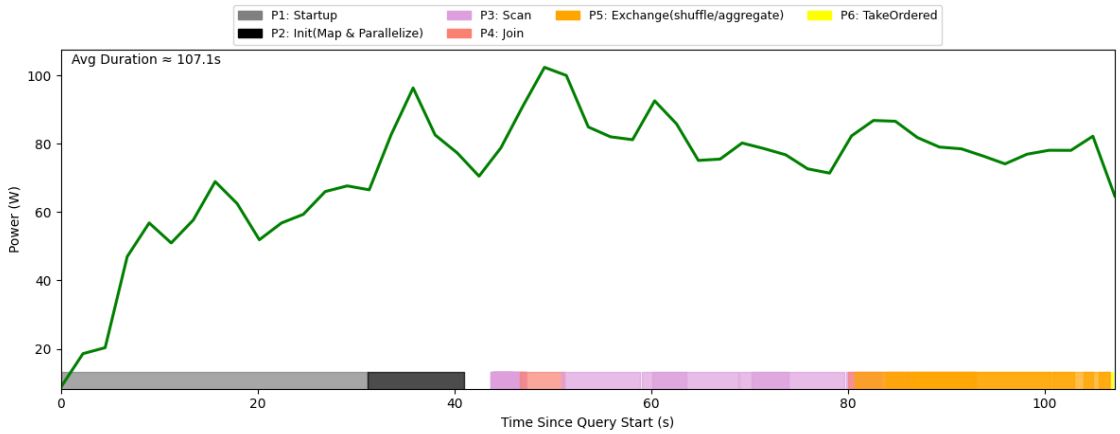
6.6.2 Power Behavior over Time

Power traces (Figures 6.29 and Figures 6.32) further confirm this interpretation:

- **q3 and q18** show smooth and moderate power curves (mostly <100W) with short, consistent durations (≈47s and ≈51s).
- **q5 and q64** display higher peak power (often >100W), longer durations (≈91s and ≈107s), and highly jagged power profiles. These fluctuations align with frequent



(a) T5-q18



(b) T5-q64

Figure 6.32: Power over time with phase highlighting: T5-q18 vs. T5-q64.

query stage transitions—particularly `Exchange` and `Aggregate` phases.

This means that complex workloads not only cost more energy in Aggregate but also create thermal and power management challenges due to sustained and volatile power usage patterns.

6.6.3 Insights and Practical Takeaways

Even when compute resources are held constant, query structure alone can lead to over 300% variation in energy usage. Lightweight scan-oriented queries like `q18` are inherently

more predictable and efficient. In contrast, queries with wide joins and complex aggregations, such as `q5` and `q64`, become disproportionately energy-intensive—not because of runtime alone, but due to data movement and coordination overheads.

To mitigate these effects, engineers should not rely solely on hardware scaling or cluster tuning. Instead, query-level optimizations can yield significant energy benefits. These include reducing shuffle width, applying early filters to minimize intermediate volume, using broadcast joins when applicable, and enabling adaptive query plans to avoid overprovisioning during execution.

From a system design perspective, this suggests that energy-aware planning must be built into query optimizers, particularly in shared or cloud-hosted analytics platforms. Developers and operators alike need to consider not just how fast a query runs but how efficiently it uses power to do so.

6.7 Threat to Validity

While we took several precautions to ensure measurement accuracy, a few limitations are worth noting.

Our experiments were conducted on virtual machines generated via the Continuum framework, which ran on a shared physical host. Although we avoided scheduling overlapping jobs, other VMs—either from our setup or from unrelated users—may have introduced background noise. This is especially relevant for power measurements taken with Scaphandre, which reports the total energy usage of the host, not individual VMs. Even lightweight activity from monitoring services or SSH sessions could have slightly skewed the results, particularly for short or low-power workloads.

To reduce this effect, we ensured that all non-essential VMs were shut down during key experiments, and the experiment controller script enforced a minimum of 95% host CPU idleness before launching the next run. Idle baselines were also executed on clean VMs with Spark fully disabled. Still, some fluctuations in idle power were observed, likely due to the nature of virtualized environments and the cumulative nature of host-level metrics reported by Scaphandre.

Another limitation relates to the way we attributed power to individual query stages. Since Scaphandre does not offer container- or VM-level breakdowns, we aligned timestamped application logs with host-level energy deltas to estimate the energy consumption per stage. These estimates were further refined using the CPU usage proportion of each VM at runtime. While this method provides reasonably accurate approximations, it is

less precise for very short or overlapping execution phases, where power changes are more volatile, and attribution is more ambiguous.

We did not benchmark alternative power measurement tools, such as PowerAPI, nor did we explore different configurations for Scaphandre startup or file-sharing mechanisms, like switching from virtiofsd to 9p[1]. The current setup focused on stability and reproducibility across runs, but it is possible that other measurement setups could have affected precision or consistency in small ways.

Lastly, all VMs in our experiments used identical resource configurations, including the same number of cores and memory. This choice helped ensure fair comparisons between configurations, but it limits generalization to more heterogeneous production environments. Our experiments were also constrained by limited hardware capacity, preventing us from scaling up to larger datasets or more complex cluster topologies. Nevertheless, the observed patterns offer a useful insight into the energy behavior of representative workloads.

[1] 9p is a lightweight file-sharing protocol originally developed for the Plan 9 operating system. It is used in virtualized environments to share files between host and guest with minimal overhead.

7

Related Work

To contextualize our work within the broader field of energy-aware big data processing, this chapter reviews relevant literature across three areas that underpin the design of our framework. We first examine how energy is monitored in virtualized and containerized environments, then turn to studies of Spark's energy behavior, and finally highlight gaps in current benchmarking methodologies. These strands of research inform both the technical underpinnings and the practical motivation for our proposed Spark-on-Kubernetes energy benchmarking pipeline.

7.1 Energy Monitoring Techniques in Virtualized and Containerized Environments

Measuring energy consumption is fundamental to evaluating the efficiency of big data systems. Among the most commonly used tools are on-chip measurement interfaces, such as Intel's Running Average Power Limit (RAPL) counters, which offer high temporal resolution and easy integration into a range of workloads (27). These tools have been widely adopted in both academic research and industry.

However, when it comes to virtualized or container-based environments, the situation becomes less straightforward. Khargharia et al. (20) proposed a hierarchical model to capture energy usage across hardware, operating system, and application layers. However, in multi-tenant cloud settings, the model's complexity increases substantially. Other studies highlight that in virtualized environments, attributing power usage to individual VMs or containers remains a significant challenge due to the shared hardware and abstraction layers at the OS level (4).

To address these difficulties, a range of lightweight energy monitoring tools have been introduced, including PowerAPI, Intel Power Gadget, and Scaphandre. While some of these tools were initially designed for physical machines or require privileged system access, recent developments have improved their applicability to virtualized and containerized settings. Among them, Scaphandre stands out for its support of Prometheus-compatible exporters and its ability to function in KVM/QEMU environments, making it a promising candidate for non-intrusive monitoring in modern cloud-native workloads. These tools provide the technical foundation for our reproducible benchmarking pipeline, which builds on Scaphandre to capture energy metrics within Spark-on-Kubernetes clusters.

7.2 Energy Behavior of Spark Workloads

The energy performance of big data frameworks, such as Hadoop and Spark, has received considerable attention over the years. Spark, in particular, benefits from its in-memory processing capabilities, which can reduce energy consumption under certain types of workloads when compared to Hadoop (10). Building on this, further studies have explored how job characteristics, resource allocation policies, and scheduling strategies influence Spark's energy footprint (26).

However, a notable gap is that much of this work centers on traditional deployment modes such as standalone or YARN. Spark-on-Kubernetes, though increasingly adopted in modern infrastructures, has seen limited investigation in this context. While there are efforts to evaluate Spark's energy use in virtualized setups (24), these often lack structured benchmarks and detailed phase-level breakdowns. Tools like HiBench and BigDataBench offer general-purpose benchmarks but typically fall short in providing automation or tracking energy use across different stages of workload execution in containerized environments.

7.3 Gaps in Reproducible Benchmarking Frameworks

Despite increasing interest in optimizing Spark for better energy efficiency, many existing studies lack a strong emphasis on reproducibility and transparency. For example, Jokanovic et al. (19) highlight how variations in access patterns can affect energy use, a factor that is frequently overlooked. Other studies also tend to neglect system-level context—such as idle power baselines, thermal throttling behavior, or CPU quota constraints—which can significantly affect measurement accuracy.

Recent work also highlights a shortage of modular benchmarking pipelines capable of profiling energy consumption across various workload stages, including data generation, training, and inference. Many evaluations are either performed manually or rely on hardware-specific setups, making it challenging to reproduce experiments or compare results across different platforms (19).

Our research addresses these challenges by developing an automated, modular benchmarking pipeline designed explicitly for Spark-on-Kubernetes environments. The framework includes synchronized power monitoring, idle baseline detection, and per-phase workload analysis. By integrating Prometheus with Scaphandre, we provide a lightweight and extensible solution for energy benchmarking that works across diverse deployment settings and facilitates reproducible experiments.

8

Lessons Learned

Throughout the development of the energy measurement framework, several unexpected technical challenges emerged—many of which could have been mitigated with more foresight, scope definition, or better tooling decisions. This section summarizes key lessons learned during the setup, deployment, and debugging of the Spark-on-Kubernetes benchmarking environment.

8.1 Spark Environment Configuration is Nontrivial and Error-prone

One of the most time-consuming components of the entire framework was configuring the Spark runtime environment on Kubernetes. The process involved building custom Docker images, configuring authentication tokens, setting up service accounts, manually importing CA certificates, and establishing secure communication between the Spark client and the Kubernetes API server. Despite Spark's support for Kubernetes, numerous edge cases required careful manual intervention.

Errors such as authorization failures, certificate mismatches, and misconfigured resource requests were common. For instance, the job submission process would silently fail if the Spark driver could not authenticate with the API server due to missing or outdated certificate authorities. Solving this required manual extraction and installation of the CA certificate into the Java trust store on the submission node—a step not documented clearly in standard Spark guides.

While the final automation pipeline encapsulated these steps reliably, achieving a working configuration required extensive trial and error. The experience highlights how container orchestration support in Spark remains nontrivial and can become a bottleneck unless

every infrastructure step is validated beforehand. Future work would benefit from building templated scripts and reducing dependency on manual steps to streamline this setup phase.

8.2 Misassumptions About Scaphandre's Capabilities Delayed Progress

Another major source of delay stemmed from an early misassumption: that Scaphandre could directly measure the power consumption of virtual machines from within the VM itself. Initial efforts focused on installing and running Scaphandre in Prometheus-exporter mode inside each guest VM. However, the resulting metrics remained static or zero, leading to confusion and wasted debugging time.

In parallel, further complications arose when attempting to synchronize Scaphandre versions across the host and VM. The expected version (v1.0.0) could not be installed using the official '.deb' packages—despite explicitly downloading versioned URLs, the installer silently deployed outdated binaries (e.g., v0.0.5). The issue was only resolved by compiling Scaphandre from the source manually, which further added to the setup complexity.

Eventually, after consulting official documentation and community forums, it became clear that Scaphandre's most reliable usage in virtualized environments is via its 'qemu' exporter, running on the host. This mode captures per-VM telemetry by monitoring virtual energy files associated with each virtual machine (VM). Coupled with Prometheus power readings from the host and proportional CPU usage, the framework could then approximate per-VM power over time with acceptable accuracy.

8.3 Continuum VM Failures Require Full Rebuilds—Robust Initialization is Crucial

A structural limitation of the Continuum framework is its dependence on QEMU-based virtual machines, which cannot be incrementally recovered after crash events. This limitation became especially problematic during large-scale experiments involving heavy Spark workloads. When resource configurations exceeded available memory (e.g., oversized datasets or under-provisioned executors), virtual machines would frequently crash or hang. Unlike cloud-based environments with recovery mechanisms, Continuum offers no rollback or snapshot support.

As a result, every VM crash necessitated a full rebuild of the affected node, followed by manual reinstallation and reconfiguration of Spark, certificates, data generators, and

monitoring tools. These repetitive tasks consumed considerable time and introduced inconsistency risks between runs.

To mitigate this, a reliable, script-driven reinitialization process was developed. It automated the restoration of Spark binaries, Docker containers, CA certificates, and experimental scripts. This tooling proved essential for restoring experimental consistency, but it also exposed a structural weakness in the infrastructure design: the lack of fault-tolerant virtual machine (VM) provisioning.

Going forward, experiments built on fragile virtual infrastructures should prioritize scriptable initialization and strict resource controls to avoid catastrophic failures. Even better, adopting infrastructure snapshots or containerized workloads with persistent states would offer significantly more resilience and repeatability.

9

Conclusion

This chapter synthesizes the findings from Chapter 5 and Chapter 6 to answer the research questions posed in Section 1.3. Drawing from both the SparkPi validation and TPC-DS benchmark experiments, we reflect on the framework's capability, its experimental coverage, and the practical takeaways for sustainable system design.

9.1 RQ1: How can a scalable and reproducible framework be designed to measure the energy consumption of Spark-based big data workloads running on Kubernetes?

To address RQ1, we developed a modular and extensible framework that integrates a QEMU-based VM orchestration layer (Continuum), a dual-mode energy telemetry system using Scaphandre, and a fully automated pipeline for task execution and measurement. This framework supports scripted cluster reset, idle detection, task submission, and post-processing—ensuring reproducibility across heterogeneous workloads and configurations.

Energy data is collected at the host level using Scaphandre's 'qemu' and 'Prometheus' exporters. Since Scaphandre cannot natively report per-VM energy usage, we approximate VM-level power using proportional CPU usage and align it with stage-level Spark events. Although indirect, this estimation method consistently reveals temporal and aggregate energy trends, allowing for phase-level attribution.

The SparkPi experiments validated this approach by demonstrating stable and interpretable energy curves across five types of scaling scenarios, confirming the pipeline's responsiveness to different resource patterns and its tolerance to noise and runtime variance. Building on this foundation, the TPC-DS experiments stress-tested the framework under

realistic and heterogeneous query workloads. These workloads introduced multi-stage execution plans, shuffle-intensive joins, and varying memory footprints, allowing us to evaluate the framework's accuracy, resolution, and generalizability.

By offering automated deployment, per-phase attribution, and compatibility with realistic workloads, this framework directly addresses the main challenge identified in this thesis: the lack of a robust, modular, and reproducible framework for measuring energy consumption in containerized, virtualized Spark environments. It provides an operational foundation for future work on energy-aware system design, workload planning, and benchmarking under repeatable and extensible conditions.

9.2 RQ2: What benchmark workloads and configuration parameters can be selected to meaningfully characterize the diversity of energy behaviors in such systems?

We addressed RQ2 by combining two levels of benchmark design: scaling configurations and workload structures, to provide an initial characterization of energy behavior diversity.

On the configuration side, we designed four types of scaling scenarios for the TPC-DS workload, each targeting a distinct dimension of system behavior. These configurations demonstrated how parallelism, resource fragmentation, and coordination overheads impact total energy and power fluctuations. For example, Type 3 revealed how scaling out can introduce significant overhead when query logic involves shuffles or joins—e.g., a 6× increase in `Join` energy from T3-1 to T3-3. Type 4 exposed how executor fragmentation (T4-3) can increase power usage despite no runtime improvement due to the costs of synchronization and communication.

On the workload side, we compared queries `q3`, `q5`, `q18`, and `q64` under identical resource setups. These were chosen to reflect canonical analytical patterns: lightweight scans, shuffle-heavy group-bys, and multi-phase aggregations. The results showed that query structure alone—without any change in cluster setup—can cause a more than threefold difference in energy usage. For instance, `q64` reached nearly 500J while `q18` consumed just under 200J.

The most informative configuration-query combinations include:

- **T3-3 (Weak Scaling with q3)** — illustrates diminishing returns and overhead growth at scale;

- **T4-3 (Split Scaling with q3)** — reveals energy penalties of executor fragmentation;
- **T5-q64 vs. T5-q18** — demonstrates how workload structure alone determines energy footprint.

These combinations meaningfully characterize energy behavior diversity by covering multiple dimensions: execution scale, task granularity, and logical complexity. Future work could extend this characterization by incorporating a broader range of queries and configuration variations to achieve more comprehensive coverage.

9.3 RQ3: What practical insights can be derived from energy measurement experiments to support more sustainable design and deployment of data-intensive systems?

The TPC-DS experiments revealed several patterns that inform energy-aware deployment strategies:

Workload parallelization does not always translate to improved efficiency. For example, in strong scaling (T1), adding more executors sometimes led to increased energy due to underutilization and coordination overhead. Similarly, split scaling (T4) demonstrated that distributing a fixed job across more executors increased total power without yielding runtime gains.

Workload complexity is a major factor. Even with identical resources, queries like `q64` and `q5` incurred substantially higher energy due to deep aggregations and shuffle operations. This suggests that optimizing query plans—e.g., reducing intermediate data volume and enabling adaptive execution—can offer significant energy savings.

Temporal power patterns also revealed that joins and exchanges are responsible for most power fluctuations and inefficiencies. For example, the `Join` phase in T3-3 contributed nearly 40% of total energy, indicating that shuffle-heavy stages should be treated as optimization targets in green computing initiatives.

In conclusion, energy optimization should be adapted to different workload contexts. When system resources are limited, grouping multiple queries into batches can improve energy efficiency by reducing the overhead of repeated task scheduling and initialization. For fixed workloads, avoiding over-fragmentation—such as splitting jobs across too many small executors—can prevent unnecessary coordination costs. For complex queries with large joins or aggregations, energy can be significantly reduced by optimizing execution

plans to minimize shuffle width and intermediate data volume. In Spark-based systems, applying different combinations of the above strategies according to specific scenarios can effectively balance performance and energy efficiency.

9.4 Future Work

This thesis establishes a modular and reproducible benchmarking framework tailored for Spark-on-Kubernetes workloads. Future work can extend this foundation in several directions. First, the architecture can be generalized to support other data processing platforms such as Hadoop and Flink, as well as GPU-accelerated workloads that exhibit different execution patterns and energy profiles. Adapting the pipeline to these systems will require adjustments in telemetry integration and workload staging, but the core automation logic can remain consistent.

Another promising direction involves enhancing the granularity of energy attribution. While current estimations rely on host-level monitoring and CPU usage ratios, integrating pod-level telemetry would enable more precise phase-level profiling across containers. This could improve the fidelity of energy diagnostics, especially in mixed-workload clusters.

Moreover, validating the framework in production-like settings with realistic job scheduling, resource contention, and multi-tenant workloads would help assess its robustness and external applicability. Finally, future experiments could focus on controlled variable analysis—isolating specific configuration parameters under strong scaling scenarios—to quantify their direct influence on power and energy. Such experiments would provide deeper insight into the causal mechanisms behind resource-energy tradeoffs, ultimately supporting more targeted and sustainable deployment strategies.

Appendix A

Reproducibility

A.1 Abstract

This appendix describes the experimental artifacts developed in this thesis to support reproducibility. The entire benchmarking framework, including automated environment setup, workload submission, and power data collection, is publicly available via GitHub. The framework is designed to be executed in Linux-based virtualized environments and produces synchronized power-over-time plots and aggregated energy metrics. It enables researchers to replicate the results presented in Chapter 5 and Chapter 6, and to further customize the pipeline for new workloads or system configurations.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Not applicable
- **Program:** Python, Bash
- **Compilation:** None (interpreted scripts)
- **Transformations:** Not applicable
- **Binary:** No
- **Model:** None
- **Data set:** Synthetic (TPC-DS generated)
- **Run-time environment:** Ubuntu 22.04 LTS, QEMU/KVM, Kubernetes 1.28, Docker 24.0, Spark 3.4.4, Prometheus, Scaphandre
- **Hardware:** x86-64 Linux machine with virtualization support
- **Run-time state:** Single-node VM cluster
- **Execution:** Fully scripted (Bash + Python automation)

- **Metrics:** Instantaneous power, cumulative energy, task duration
- **Output:** PNG plots and TXT logs
- **Experiments:** SparkPi and TPC-DS scaling/query scenarios
- **How much disk space required (approximately)?:** 15 GB
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes (with VM rebuild)
- **How much time is needed to complete experiments (approximately)?:** 2–4 hours depending on configuration
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Data licenses (if publicly available)?:** TPC-DS license
- **Workflow framework used?:** Continuum (custom script-based)
- **Archived (provide DOI)?:** Not archived (GitHub only)

A.3 Description

A.3.1 How to access

The entire artifact is hosted on GitHub at: `https://github.com/Ellies1/thesis01`

This repository includes the benchmarking framework, installation scripts, workload definitions, and result generation code.

A.3.2 Hardware dependencies

The framework must be run on a Linux host machine with virtualization support (Intel VT-x or AMD-V), at least four physical cores, and 16 GB RAM. No specific GPU or hardware sensors are required, but `qemu` must be available.

A.3.3 Software dependencies

- Host OS: Ubuntu 22.04 LTS
- Virtualization: QEMU + KVM
- Containerization: Docker 24.0
- Orchestration: Kubernetes 1.28
- Benchmarking tools: Apache Spark 3.4.4

- Energy monitoring: Scaphandre (with qemu and Prometheus exporters)
- Monitoring: Prometheus, Python 3.10+, matplotlib

A.3.4 Data sets

The TPC-DS dataset is generated within the script using Spark's built-in data generator at 10 GB scale. No external datasets are required.

A.3.5 Models

Not applicable.

A.4 Installation

Clone the repository and follow the `README.md` instructions. The main installation entry point is:

```
bash remote-setup-every.sh
```

This script automates VM creation, Spark configuration, Kubernetes cluster setup, and energy exporter initialization. No manual installation is needed.

A.5 Experiment workflow

After installation, execute benchmark experiments using:

```
python3 TPCDSEC616/v2.py
```

This script handles the full experimental pipeline:

- Reset cluster to a clean state
- Wait for idle energy stabilization
- Submit workload to Kubernetes (SparkPi or TPC-DS queries)
- Collect and timestamp energy metrics via Scaphandre
- Generate power-over-time plots and energy summary figures

All outputs are stored in the `/output` directory in both `.png` and `.txt` formats.

A.6 Evaluation and expected results

The expected results include:

- PNG figures showing power fluctuation during different execution phases
- TXT logs recording cumulative energy per stage/query
- Validation plots for five SparkPi scenarios
- Energy comparison charts across TPC-DS configurations and queries

These outputs should match the trends and insights reported in Chapter 6. Some variability may occur due to runtime noise, but the overall energy patterns should remain reproducible.

A.7 Experiment customization

To run different queries or configurations, modify the `experiment_configs` inside `v2.py`. Users may specify:

- Number of executors
- Core/memory per executor
- TPC-DS query name
- Experiment repeat times

New workloads can be added by modifying the pipeline logic and tagging stages in the Spark logs.

A.8 Notes

The framework is primarily designed for academic research. It may require elevated privileges to run virtual machines and access `/dev/kvm`. Ensure proper kernel modules are enabled on the host.

A.9 Methodology

Submission and reviewing of reproducibility materials follow the ACM guidelines:

- `https://www.acm.org/publications/policies/artifact-review-badging`
- `http://cTuning.org/ae/submission-20201122.html`
- `http://cTuning.org/ae/reviewing-20201122.html`

References

[1] Tpc benchmark™ ds. Transaction Processing Performance Council, 2024. Available at http://www.tpc.org/tpcds/. 9, 16

[2] Abukari, A. M., Nimatu, S., and Madavarapu, J. B. Comparative analysis of virtualization and containerization: Performance perspective. Department of Computer Science, Tamale Technical University, Ghana; Department of Information Technology, University of the Cumberlands, USA. 3

[3] Apache Software Foundation. Apache spark examples: Sparkpi. https://spark.apache.org/examples.html, 2025. Accessed: 2025-05-11. 5, 9, 28

[4] Basmadjian, R., Meer, H. D., and Lent, R. Evaluation of power monitoring techniques for data centers. *IEEE Transactions on Sustainable Computing 1*, 2 (2013), 100–112. 57

[5] Buyya, R., Ilager, S., and Arroba, P. Energy-efficiency and sustainability in new generation cloud computing: A vision and directions for integrated management of data centre resources and workloads. *Software: Practice and Experience 53*, 4 (2023), 1–20. 1, 2

[6] Cloud Native Computing Foundation. Cncf research reveals how cloud native technology is reshaping global business and innovation, 2025. Available at: https://www.cncf.io/reports/. 1

[7] Freels, A. Learn how to monitor your energy use at home with a raspberry pi, grafana and prometheus. https://grafana.com/blog/2021/04/15/learn-how-to-monitor-your-energy-use-at-home-with-a-raspberry-pi-grafana-and-prometh 2021. Accessed: 2025-05-15. 3

[8] Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., and Jacobsen, H.-A. Bigbench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, Association for Computing Machinery, p. 1197–1208. 16

[9] Henning, S., and Hasselbring, W. Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud. *arXiv preprint arXiv:2303.11088* (2023). 2

[10] Huang, Y., and Buyya, R. A comparative study of energy-efficient scheduling in hadoop and spark. *Journal of Parallel and Distributed Computing 72*, 11 (2012), 1374–1386. 58

[11] Hubblo. Qemu exporter - scaphandre documentation. `https://hubblo-org.github.io/scaphandre-documentation/references/exporter-qemu.html`. Accessed: 2025-05-23. 26

[12] Hubblo. Scaphandre: Energy consumption monitoring agent, by process, with prometheus export. `https://github.com/hubblo-org/scaphandre`, 2020. Accessed: 2025-05-11. 4, 26

[13] Hubblo. Explanation on rapl / running average power limit domains: what we (think we) know so far. `https://hubblo-org.github.io/scaphandre-documentation/explanations/rapl-domains.html`, 2023. Accessed: 2025-05-22. 3, 26

[14] Hubblo. How Scaphandre computes per process power consumption, 2025. Online; accessed 2025-05-02. 3, 4, 8

[15] Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing in Science Engineering 9*, 3 (2007), 90–95. 5

[16] Intel-bigdata. Hibench: A big data benchmark suite. `https://github.com/Intel-bigdata/HiBench`. Accessed: 2025-05-23. 16

[17] International Energy Agency. AI is set to drive surging electricity demand from data centres while offering the potential to transform how the energy sector works, Apr. 2025. Online; accessed 2025-04-10. 1

[18] Jansen, M. Continuum: Experimental research automation for distributed systems. `https://github.com/atlarge-research/continuum`, 2023. Accessed: 2025-05-11. 8, 25

[19] Jokanovic, B., and Gelenbe, E. Impact of access patterns on energy consumption in big data systems. In *Proc. of IEEE BigData* (2016). 58, 59

[20] Khargharia, B., and Smith, J. E. A hierarchical energy measurement framework for distributed systems. In *Proc. of IEEE IPDPS* (2010). 57

[21] Poess, M., Nambiar, R. O., and Walrath, D. Why you should run tpc-ds: a workload analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (2007), VLDB '07, VLDB Endowment, p. 1138–1149. 34

[22] Prometheus Authors. Prometheus: Monitoring system & time series database. `https://prometheus.io/`, 2012. Accessed: 2025-05-22. 4, 26

[23] Savazzi, S., Rampa, V., Kianoush, S., and Bennis, M. An energy and carbon footprint analysis of distributed and federated learning. *arXiv preprint arXiv:2206.10380* (2022). 2

[24] Tang, Z., Chen, W., and Rao, J. Spark energy evaluation in virtualized environments. In *Proc. of ACM EuroSys* (2022). 58

[25] Vennu, V. K., and Yepuru, S. R. A performance study for autoscaling big data analytics containerized applications. Master's thesis, Blekinge Institute of Technology, 2022. iii, 2

[26] Wang, W., Liu, J., and He, T. Energy-aware scheduling in apache spark: A case study. In *Proc. of IEEE GreenCom* (2021). 58

[27] Wang, X., Wang, Y., and Pedram, M. Energy measurement and modeling for rapl in intel cpus. In *Proc. of ISLPED* (2019). 57

[28] Wang, Y., and Wang, Y. The impact of digital infrastructure on innovation and economic growth. *Journal of Innovation & Knowledge 7*, 1 (2012), 1–10. 1

[29] Wikipedia contributors. Climate Neutral Data Centre Pact, 2025. Wikipedia, The Free Encyclopedia. Online; accessed 2025-05-02. 1

[30] Zhu, C., Han, B., and Zhao, Y. A comparative performance study of spark on kubernetes. *The Journal of Supercomputing 78*, 11 (2022), 13298–13322. iii, 2, 4