Zurich University of Applied Sciences

Department School of Engineering

Institute of Computer Science

MASTER THESIS

Tycho:

An Accuracy-First Architecture for Server-Wide Energy Measurement and Process-Level Attribution in Kubernetes

---

Author:
Caspar Wackerle

Supervisors:
Prof. Dr. Thomas Bohnert
Christof Marti

Submitted on
January 31, 2026

Study program:
Computer Science, M.Sc.

Imprint

Project: Master Thesis
Title: Tycho: An Accuracy-First Architecture for Server-Wide Energy Measurement and Process-Level Attribution in Kubernetes
Author: Caspar Wackerle
Date: January 31, 2026
Keywords: process-level energy consumption, cloud, kubernetes, kepler

Study program:
Computer Science, M.Sc.
Zurich University of Applied Sciences

Supervisor 1:
Prof. Dr. Thomas Bohnert
Zurich University of Applied Sciences
Email: thomas.michael.bohnert@zhaw.ch
Web: Link

Supervisor 2:
Christof Marti
Zurich University of Applied Sciences
Email: christof.marti@zhaw.ch
Web: Link

Abstract

Abstract

The accompanying source code for this thesis, including all deployment and automation scripts, is available in the **PowerStack**[1] repository on GitHub.

Contents

Appendix A: Container-Level Energy Consumption Estimation: Foundations, Challenges, and Current Approaches

Appendix B: Implementation of an energy monitoring environment in Kubernetes

List of Figures

List of Tables

Appendix A

Appendix B

The global climate crisis is one of humanity's greatest challenges in this century.
With this work, I hope to contribute a small part in the direction we urgently need to go.

Chapter 1

Introduction

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
REVISE ENTIRE CHAPTER LATER XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

1.1 Motivation

Energy consumption in data centers continues to rise as demand for compute-intensive and latency-sensitive services increases. Modern cloud platforms host diverse workloads such as machine learning inference, analytics pipelines, and high-density microservices, all of which collectively contribute to a growing global electricity footprint. Container orchestration frameworks amplify these trends by enabling dense consolidation of workloads across shared servers. While this improves resource efficiency, it also introduces abstraction layers that obscure the relationship between workload behaviour and physical energy use.

As interest in sustainable cloud operations intensifies, there is increasing demand for precise, workload-level energy visibility. Fine-grained and reproducible energy measurements are essential for research domains such as performance engineering, scheduling, autoscaling, and the design of energy-aware systems. Existing tools provide valuable approximations but prioritise portability and low operational overhead, and therefore do not target the upper bounds of measurement fidelity. Research environments, by contrast, require methodologies that prioritise accuracy, control, and verifiability over deployability.

This thesis is motivated by the need for an accuracy-focused measurement approach that supports rigorous experimental work on containerised systems. Rather than proposing new optimisation mechanisms, this work concentrates on establishing a reliable methodological foundation for observing and analysing workload-induced energy consumption in controlled settings.

1.2 Problem Context

Modern multi-tenant servers host many short-lived and highly dynamic workloads that execute concurrently and compete for shared hardware resources. On such systems, the aggregate power draw represents the combined activity of numerous interacting subsystems, while the contributions of individual workloads remain deeply entangled. Containerisation further complicates this picture: processes belong to containers, containers belong to pods, and pods may change state rapidly under

orchestration. These abstractions improve system management but obscure how computational activity translates into power consumption.

At the same time, servers expose a heterogeneous collection of telemetry sources. Each source reflects different aspects of hardware behaviour, updates at its own cadence, and provides only a partial view of system activity. Because workload state changes and telemetry updates occur independently, they do not naturally align in time. The resulting temporal misalignment limits the reliability of workload-level energy attribution and leads to uncertainty in short-duration or phase-sensitive analyses.

Kubernetes introduces additional challenges. Workloads may start and terminate within milliseconds, metadata may appear with delays, and lifecycle events may interleave in complex ways. Existing tools often rely on coarse sampling windows or heuristic models that mask these inconsistencies. While sufficient for operational monitoring, such abstractions constrain the achievable accuracy in research settings. An accuracy-oriented approach requires explicit treatment of timing, metadata consistency, and correlation across heterogeneous measurement sources.

1.3 Position Within Previous Research

This thesis builds upon two earlier stages of work. The implementation-focused VT1 project developed an initial measurement pipeline and explored practical aspects of collecting hardware and system-level metrics in a Kubernetes environment. The subsequent VT2 project examined the state of the art in server-level energy measurement, validated the behaviour of commonly used telemetry sources, and identified methodological and technical limitations in existing tools such as Kepler. Both works are included in the appendix as supporting material.

The present thesis integrates these earlier insights but does not repeat them. Instead, it synthesises the essential findings from VT2 in a condensed form (Chapter 2), and introduces the conceptual foundations required to reason about accurate energy attribution (Chapter 3). These chapters provide the background necessary to understand the accuracy-focused architecture developed later in this thesis.

1.4 Problem Statement

Accurately determining how much energy individual workloads consume in a Kubernetes cluster remains a challenging open problem. Clusters host many short-lived and overlapping workloads whose behaviour evolves rapidly, while server-level power telemetry is exposed through heterogeneous interfaces that update asynchronously and lack consistent timestamps. These timing mismatches, combined with the abstraction layers introduced by container orchestration, obscure the relationship between workload activity and physical energy use. Existing approaches provide high-level estimates but cannot deliver the temporal alignment, attribution fidelity, or reproducibility required for rigorous experimental analysis. This thesis therefore addresses the problem of designing a measurement methodology and prototype system capable of producing time-aligned, workload-level energy attribution with sufficient accuracy for research environments.

1.5 Goals of This Thesis

The overarching goal of this thesis is to develop an accuracy-focused approach for measuring energy consumption in Kubernetes-based environments. To achieve this, the work pursues four concrete objectives:

- **Methodological objective:** Define a measurement methodology that aligns heterogeneous telemetry sources with dynamic workload behaviour under a unified temporal model suitable for controlled research settings.
- **Architectural objective:** Design an accuracy-first system architecture that explicitly handles timing, metadata consistency, and correlation across diverse metrics without relying on heuristic abstractions.
- **Prototype objective:** Implement a research prototype that realises this architecture on commodity server hardware and integrates workload metadata, timing information, and server-wide telemetry into a coherent measurement pipeline.
- **Foundational objective for future work:** Establish the methodological and architectural basis for subsequent validation studies that will evaluate measurement fidelity and explore trade-offs between accuracy, overhead, and operational constraints.

1.6 Research Questions

1. How reliably can an accuracy-focused measurement approach capture and represent workload-induced variations in energy consumption within dynamic, multi-tenant Kubernetes environments?
2. To what extent does a unified timing and attribution methodology improve the consistency and interpretability of workload-level energy measurements compared to existing estimation-oriented approaches?
3. In which contexts does high-fidelity energy measurement provide meaningful benefits for research and experimental analysis, and what trade-offs arise between accuracy, overhead, and operational constraints?

1.7 Contributions

This thesis makes several conceptual and methodological contributions to the study of energy measurement in container-orchestrated environments. First, it introduces an accuracy-focused measurement approach that prioritizes temporal consistency, reproducibility, and the faithful representation of workload behaviour. The work defines a methodology for unifying heterogeneous sources of server telemetry under a shared timing model, enabling coherent interpretation of workload activity and system-level energy use.

A second contribution is the development of a prototype system that operationalizes this methodology and provides a concrete platform for exploring the limits of

high-fidelity energy measurement in Kubernetes-based environments. The prototype integrates workload metadata, timing information, and server-wide telemetry into a coherent measurement pipeline designed for research and controlled experimentation.

Third, the thesis establishes a foundation for reliable workload-level attribution by describing a structured process for correlating dynamic workload behaviour with system energy consumption. This provides a basis for analysing short-lived workload phases, transient resource usage patterns, and other phenomena that require fine-grained temporal alignment.

Finally, the work prepares the methodological groundwork for subsequent validation studies by outlining experimental procedures, calibration strategies, and evaluation principles suited to accuracy-oriented measurement. Together, these contributions advance the methodological state of the art and offer a practical reference point for future research on energy transparency in modern cloud infrastructures.

1.8 Scope and Boundaries

This thesis focuses on high-level principles and methods for energy measurement in multi-tenant server environments. The primary scope includes conceptual design, prototype development, and preparation of the methodological foundation for subsequent evaluation work. The emphasis is on accuracy, reproducibility, and consistency rather than operational deployability or production-grade integration.

Several areas remain outside the scope of this work. The thesis does not propose scheduling policies, predictive models, or system-level optimisation mechanisms. It does not modify Kubernetes or introduce changes to cloud operators' workflows. The prototype developed in this thesis is intended for controlled research environments and does not aim to provide a turnkey solution for general-purpose use. The work assumes access to a server environment where low-level telemetry and measurement interfaces are accessible under suitable conditions.

1.9 Origin of the Name "Tycho"

The prototype developed in this thesis is named *Tycho*, a reference to the astronomer Tycho Brahe. Brahe is known for producing exceptionally precise astronomical measurements, which later enabled Johannes Kepler to formulate the laws of planetary motion. The naming reflects a similar relationship: while the upstream *Kepler* project focuses on modelling and estimation, this thesis explores the upper bounds of measurement accuracy. Tycho thus signals both continuity with prior work and a shift toward an accuracy-first design philosophy.

1.10 Methodological Approach

1.11 Thesis Structure

Chapter 2

Background and Related Research

This chapter summarises the current state of research and industrial knowledge on server-level energy measurement. Its focus is limited to what the literature reports about available telemetry sources, measurement techniques, and existing attribution tools. The discussion is descriptive rather than conceptual: it does not introduce attribution principles, methodological reasoning, or design considerations, which are addressed in Chapter 3. Extended background material is available in Appendix A and the present chapter integrates only those findings that are directly relevant for understanding the research landscape.

2.1 Energy Measurement in Modern Server Systems

The energy consumption of modern servers arises from a heterogeneous set of subsystems, including CPUs, GPUs, memory, storage devices, network interfaces, and platform management components. Prior research highlights that these subsystems expose highly unequal visibility into their power behaviour, since measurement capabilities, granularity, and accuracy differ significantly across hardware generations and vendors [2, 3]. Some domains provide direct telemetry, while others can only be approximated through software-derived activity metrics. As a result, no single interface offers complete or temporally consistent power information, and most studies rely on a single source or combine multiple sources to approximate system-level consumption. This fragmented measurement landscape forms the basis for much of the existing work on power modelling, validation, and multi-source energy estimation in server environments.

2.1.1 Energy Attribution in Multi-Tenant Environments

Several studies identify containerised and multi-tenant systems as challenging environments for energy attribution. Containers share the host kernel and rely on common processor, memory, storage, and network subsystems, which removes the isolation boundaries present in virtual machines and prevents direct measurement of per-container power. Research reports that workloads running concurrently on the same node create interference effects across hardware domains, leading to utilisation patterns that correlate only loosely with actual energy consumption [2]. Modern orchestration platforms further increase attribution difficulty through highly dynamic execution behaviour: containers are created, destroyed, and rescheduled at high frequency, often numbering in the thousands on large clusters. These rapid lifecycle changes produce volatile metadata and short-lived resource traces that are difficult

to align with node-level telemetry. Collectively, the literature treats container-level energy attribution as an estimation problem constrained by incomplete observability, heterogeneous measurement quality, and continuous runtime churn.

2.1.2 Telemetry Layers in Contemporary Architectures

Modern servers expose power and activity information through two largely independent telemetry layers. The first consists of in-band mechanisms that are visible to the operating system, including on-die energy counters, GPU management interfaces, and kernel-level resource statistics. These interfaces typically offer higher sampling rates and finer granularity, but their accuracy and coverage vary across hardware generations and vendors. Prior work notes that in-band telemetry often represents estimated rather than directly measured power and that several domains, such as network and storage devices, expose only partial or indirect information.

The second layer is out-of-band telemetry provided by baseboard management controllers through interfaces such as IPMI or Redfish. These systems aggregate sensor readings independently of the host and report stable, whole-system power values at coarse temporal resolution. Empirical studies show that out-of-band telemetry provides useful system-level accuracy, although update intervals and measurement precision differ substantially between vendors [4]. Compared with instrument-based measurements, which remain the benchmark for high-fidelity evaluation but are impractical at scale, both in-band and out-of-band methods represent trade-offs between granularity, availability, and measurement reliability.

Combined, these layers form a heterogeneous telemetry landscape in which sampling rates, accuracy, and domain coverage differ significantly, motivating the use of multi-source measurement approaches in research.

2.1.3 Challenges for Container-Level Measurement

Existing research identifies several factors that complicate accurate energy measurement for containerised workloads. Large-scale trace analyses show that cloud environments exhibit substantial churn, with many tasks being short-lived and resource demands changing rapidly over time [5]. Such dynamism limits the observability of fine-grained resource usage and makes it difficult to capture short execution intervals with sufficient temporal resolution.

Monitoring studies further report inconsistencies across the different layers that expose resource information for containers. In multi-cloud settings, observability often depends on heterogeneous monitoring stacks, leading to fragmented visibility and non-uniform coverage of system activity [6]. Even within a single host, performance counters obtained from container-level interfaces may diverge from system-level measurements. Empirical evaluations demonstrate that container-level CPU and I/O counters can underestimate actual activity by a non-negligible margin, and that co-located workloads introduce contention effects that distort these metrics [7].

These findings indicate that container-level measurement operates under conditions of rapid workload turnover, heterogeneous monitoring behaviour, and imperfect resource visibility. As a consequence, the literature treats container energy attribution as a problem constrained by incomplete and potentially biased measurement signals rather than as a directly measurable quantity.

2.2 Hardware and Software Telemetry Sources

This section outlines the primary telemetry sources used to observe power and resource behaviour in modern server systems. It summarises established research on external measurement devices, firmware-level interfaces, on-die energy counters, accelerator telemetry, and kernel-exposed resource metrics. The emphasis is on reporting the properties and empirical characteristics documented in prior work, without interpreting these signals conceptually or analysing their temporal behaviour, which are addressed in later sections. A comprehensive technical discussion is provided in Appendix A, Chapter 2; the present section extracts only the findings relevant for understanding the measurement landscape.

2.2.1 Direct Hardware Measurement

Direct physical instrumentation remains the most accurate method for measuring server power consumption. External power meters or inline shunt-based devices can capture node-level energy usage with high fidelity, and research frequently uses such instrumentation as a ground truth for validating software-reported power values. Studies employing dedicated measurement setups, such as custom DIMM-level sensing boards, demonstrate that high-frequency sampling and component-level granularity are technically feasible but require bespoke hardware and non-trivial integration effort [8]. Lin et al. classify these approaches as offering very high data credibility but only coarse spatial granularity and limited scalability in operational environments [2].

Recent work on specialised sensors, such as the PowerSensor3 platform[9] for high-rate voltage and current monitoring of GPUs and other accelerators, illustrates ongoing interest in hardware-centric power measurement. However, these systems share the same fundamental drawback: deployment across production servers is complex, costly, and incompatible with large-scale or multi-tenant settings. As a consequence, direct instrumentation is predominantly used in controlled experiments or for validation of other telemetry sources, rather than as a primary measurement mechanism in real-world server infrastructures.

2.2.2 Legacy Telemetry Interfaces (ACPI, IPMI)

Early power-related telemetry on server platforms was primarily exposed through ACPI and IPMI. ACPI provides a standardised interface for configuring and controlling hardware power states, but it does not offer real-time energy or power readings. The interface exposes only abstract performance and idle states defined by the firmware [10], and these states do not include the instantaneous power information required for empirical energy measurement. Consequently, ACPI has seen little use in modern power estimation research.

IPMI, accessed through the baseboard management controller, represents an older class of out-of-band telemetry that predates Redfish. Although widely supported across server hardware, IPMI power values are known to be coarse, slowly refreshed, and often inaccurate when compared with external instrumentation. Empirical studies report multi-second averaging windows, substantial quantisation effects, and unreliable idle power readings [11, 12]. These limitations, together with the availability of more precise alternatives, have led IPMI to be largely superseded by Redfish on contemporary server platforms.

2.2.3 Redfish Power Telemetry

Redfish is the modern out-of-band management interface available on contemporary server platforms and is designed as the successor to IPMI. It exposes system-level telemetry through a RESTful API implemented on the baseboard management controller (BMC), providing access to whole-node power readings derived from on-board sensors. Prior work consistently shows that Redfish delivers higher precision than IPMI, with lower quantisation artefacts and more stable readings across power ranges [4]. In controlled experiments, Redfish achieved a mean absolute percentage error of roughly three percent when compared to a high-accuracy power analyser, outperforming IPMI in all evaluated power intervals.

A key limitation of Redfish is its temporal granularity. Empirical studies report that power values exhibit non-negligible staleness, with refresh delays of approximately 200 ms [4]. This latency restricts the ability of Redfish to capture short bursts of activity or rapid fluctuations in dynamic workloads. Accuracy and responsiveness also vary across vendors, reflecting differences in embedded sensors, BMC firmware, and management controller architectures.

The interface is widely deployed in real-world infrastructure. Modern enterprise servers from Dell, HPE, Lenovo, Cisco, and Supermicro routinely expose power telemetry via Redfish as part of their standard BMC firmware [13]. Out-of-band monitoring studies further highlight that Redfish avoids the overheads and failure modes associated with in-band agents [14]. In practice, Redfish implementations tend to provide stable low-frequency updates suitable for coarse-grained power reporting.

Preliminary measurements conducted for this thesis also observed irregular update intervals on the evaluated hardware, occasionally extending into the multi-second range. While this behaviour is specific to a single system and not generalisable, it reinforces the literature's position that Redfish telemetry exhibits meaningful vendor-dependent variability and remains unsuitable for fine-grained temporal correlation.

Overall, Redfish provides accessible, reliable whole-node power telemetry at coarse temporal resolutions, making it valuable for long-interval monitoring and for validating other measurement sources, but inappropriate for attributing energy consumption to short-lived or rapidly fluctuating containerised workloads.

2.2.4 RAPL Power Domains

Running Average Power Limit (RAPL) provides hardware-backed energy counters for several internal power domains of a processor package. Originally introduced by Intel and later adopted in a compatible form by AMD, RAPL exposes energy measurements via model-specific registers that can be accessed directly or through higher-level interfaces such as the Linux `powercap` framework or the `perf-events` subsystem [15, 16]. Raffin et al. provide a detailed comparison of these access mechanisms, noting that MSR, powercap, perf-events, and eBPF differ mainly in convenience, required privileges, and robustness; all can retrieve equivalent RAPL readings when implemented correctly [16]. They recommend accessing RAPL via the powercap interface, which is easiest to implement reliably and suffers from no overhead penalties when compared with more low-level methods.

Intel platforms typically expose several well-established RAPL domains, including the processor package, the core subsystem, and (on many server architectures) a DRAM domain [17]. These domains have been validated extensively against external measurement equipment. Studies report that the combination of package and DRAM energy tracks CPU-and-memory power with good accuracy from Haswell onwards, which has led to RAPL becoming the primary fine-grained energy source in server-oriented research [8, 18–20]. More recent work on hybrid architectures such as Alder Lake confirms that RAPL continues to correlate well with external measurements under load, while precision decreases somewhat in low-power regimes [21]. Across these studies, RAPL is generally regarded as sufficiently accurate for scientific analysis when its domain boundaries and update characteristics are considered [16].

AMD implements a RAPL-compatible interface with a similar programming model but a reduced set of domains. Zen 1 through Zen 4 processors expose package and core domains only, without a dedicated DRAM domain [16, 22]. Schöne et al. show that, as a consequence, memory-related energy may not be represented explicitly in AMD's RAPL output, leading to a smaller portion of total system energy being observable through the package domain alone [22]. This limitation primarily concerns domain completeness rather than measurement correctness: for compute-intensive workloads, package-domain values behave consistently, but workloads with significant memory activity exhibit a larger gap relative to whole-system measurements because DRAM energy is not separately reported. Raffin et al. further note that, on the evaluated Zen-based server, different kernel interfaces initially exposed inconsistent domain sets; this was later corrected upstream, illustrating that AMD support is evolving and still maturing within the Linux ecosystem [16].

Technical considerations also apply to both Intel and AMD platforms. RAPL counters have finite width and wrap after sufficiently large energy accumulation, requiring consumers to implement overflow correction [16, 23]. The counters do not include timestamps, and empirical work shows that actual update intervals may deviate from nominal values, complicating precise temporal correlation with other telemetry [18, 24]. On some Intel platforms, security hardening measures such as energy filtering reduce temporal granularity for certain domains to mitigate side-channel risks [21, 25, 26]. In virtualised environments, RAPL access may be trapped by the hypervisor, increasing latency and introducing small deviations from bare-metal behaviour [24].

In summary, RAPL provides a widely used and comparatively fine-grained source of processor-side energy telemetry. Intel platforms typically offer multiple validated domains, including DRAM, enabling a broader view of CPU-and-memory energy. AMD platforms expose fewer domains and therefore provide a more limited perspective on total system power, particularly for memory-intensive workloads. These differences in domain coverage, measurement scope, and software integration need to be taken into account when using RAPL as a basis for energy analysis.

2.2.5 GPU Telemetry

Unlike CPUs, where power and utilization telemetry is supported through standardised interfaces, GPU energy visibility relies primarily on vendor-specific mechanisms. For NVIDIA devices, two interfaces dominate this landscape: the *NVIDIA Management Library* (NVML), which has become the industry standard, and the *Data*

Center GPU Manager (DCGM), a less widely used management layer that also exposes telemetry.

2.2.5.1 NVML

NVML is NVIDIA's primary interface for device-level monitoring and underpins tools such as `nvidia-smi`. It provides access to power, energy (on selected data-center GPUs), GPU utilization, memory usage, clock frequencies, thermal state, and various health and throttle indicators. Among these, power and utilization are most relevant for energy analysis.

NVML power values represent board-level estimates derived from on-device sensing circuits and are shaped by internal averaging and architecture-dependent update behaviour. Recent empirical studies across modern devices show that NVML produces fresh samples only intermittently and applies smoothing that reduces the visibility of short-lived power changes, while steady-state power levels remain comparatively accurate [27]. On the Grace–Hopper GH200, these effects are pronounced: NVML reflects a coarse internal averaging interval and therefore underrepresents short kernels and transient peaks relative to higher-frequency system interfaces [28]. These findings indicate that NVML captures long-term power behaviour reliably but inherently limits fine-grained visibility. Despite these constraints, existing studies consistently find that NVML provides reasonably accurate steady-state power estimates on modern data-center GPUs and currently represents the most reliable and widely supported mechanism for obtaining GPU power telemetry in practical systems [28].

GPU utilization provides contextual information about device activity. It reports the proportion of time during which the GPU is executing any workload rather than the fraction of computational capacity in use, making it a coarse activity indicator rather than a detailed performance metric [29].

2.2.5.2 DCGM

DCGM is NVIDIA's management and observability framework designed for data-center deployments. It aggregates telemetry, performs health monitoring, exposes thermal and throttle state, and provides detailed visibility in environments that employ Multi-Instance GPU (MIG) partitioning. However, DCGM's power and utilization metrics are derived from the same underlying measurement sources as NVML. In practice, DCGM is far less commonly used for energy analysis because it does not provide higher-fidelity power telemetry; instead, it applies additional aggregation and is typically deployed with coarse sampling intervals, especially when used through exporters in cluster monitoring systems. DCGM therefore represents an alternative access path to the same measurements rather than a distinct source of energy-related information.

DCGM is considerably less common in both research and operational practice, with most GPU monitoring systems relying primarily on NVML while DCGM appears only occasionally in cluster-level deployments [29].

2.2.5.3 Summary

NVML and DCGM jointly define the available mechanisms for GPU telemetry in cloud environments. NVML is the dominant and broadly supported interface for power and utilization measurement, while DCGM extends it with operational metadata and management integration. Current studies consistently show that both interfaces expose averaged, device-level power estimates that capture long-term behaviour but are inherently limited in their ability to represent short-duration activity or fine-grained workload structure. These characteristics form the scientific foundation for later discussions of temporal behaviour and measurement methodology.

2.2.6 Software-Exposed Resource Metrics

In addition to hardware telemetry, Linux and Kubernetes expose a wide range of software-level resource metrics that describe system and workload activity. These metrics do not measure power directly but provide essential behavioural context that complements RAPL, Redfish, and GPU telemetry.

2.2.6.1 CPU and Memory Activity Metrics

Linux provides several complementary mechanisms for tracking CPU and memory usage. Global counters such as `/proc/stat` record cumulative CPU time since boot, while per-task statistics in `/proc/<pid>` expose user-mode and kernel-mode execution time with high granularity [30]. Control groups (cgroups) provide container-level CPU and memory accounting and form the primary basis for utilisation metrics inside Kubernetes [31, 32]. Higher-level tools such as cAdvisor and metrics-server aggregate this information via Kubelet, but at significantly lower update rates.

Event-driven approaches provide substantially finer resolution. eBPF allows dynamic attachment to kernel events such as context switches, scheduling decisions, and I/O operations, enabling near-real-time capture of per-task CPU activity with low overhead [33, 34]. Hardware performance counters accessed through `perf` offer insight into instruction counts, cycles, cache behaviour, and stalls [35]. These sources provide detailed behavioural information but still represent utilisation rather than energy.

2.2.6.2 Storage Activity Metrics

Storage subsystems do not expose real-time power telemetry, yet Linux provides a rich set of activity indicators. Per-process statistics in `/proc/<pid>/io` track bytes read and written, while cgroup I/O controllers report aggregated container-level metrics. Subsystem-specific tools such as `smartctl` and `nvme-cli` reveal additional device characteristics, queue behaviour, and state transitions [36, 37].

In the absence of hardware power sensors, multiple works propose workload-dependent energy models for storage devices [38–40]. These models can yield accurate estimates when calibrated for a specific device but do not generalise across heterogeneous hardware due to differences in flash controllers, firmware, and internal data paths.

2.2.6.3 Network and PCIe Device Metrics

Network interfaces provide byte and packet counters via `/proc/net/dev`, but expose no dedicated power telemetry. Research models for NIC energy consumption exist [41–43], yet all rely on device-specific idle and active power characteristics that are not available at runtime. Similarly, PCIe devices support abstract power states as defined by the PCIe specification [44], but these states do not reflect instantaneous power usage and thus offer only coarse activity signals.

2.2.6.4 Secondary System Components

Components such as fans, motherboard logic, and power delivery subsystems rarely expose fine-grained telemetry. Although some BMC implementations report coarse sensor values, these readings are inconsistent across platforms and generally unsuitable for high-resolution analysis. Consequently, research commonly treats these subsystems as part of the residual power that scales with the activity of primary components [42].

2.2.6.5 Model-Based Estimation Approaches

Because software-visible metrics capture detailed workload behaviour, many works propose inferring energy consumption from utilisation using regression or stochastic models [45–48]. While these models can be effective when fitted to a specific hardware platform, their accuracy depends heavily on device-specific parameters, making them unsuitable as a general mechanism for heterogeneous server environments. Machine-learning-based estimators share the same limitation: high accuracy when trained for a fixed configuration, poor portability without extensive retraining.

2.2.6.6 Summary

Software-exposed metrics provide high-resolution visibility into CPU, memory, I/O, and network activity. They are indispensable for correlating workload behaviour with hardware power signals, especially for components that lack native telemetry. Model-based estimation remains possible but inherently platform-specific, and therefore unsuitable as a universal foundation for fine-grained attribution in heterogeneous environments.

2.3 Temporal Behaviour of Telemetry Sources

A comprehensive treatment of temporal characteristics can be found in Appendix A, Chapter 2, but the present section focuses on the empirical, source-specific behaviours that constrain fine-grained power and energy estimation on real systems. Modern server platforms expose a heterogeneous set of telemetry interfaces, and their timing properties vary substantially: some update at fixed intervals, others employ internal averaging or smoothing, several expose counters without timestamps, and many lack guarantees on refresh regularity. These behaviours shape the effective temporal resolution with which workload-induced power changes can be observed.

The purpose of this section is not to develop a conceptual theory of sampling or to explain why timing matters for attribution (both are deferred to Chapter 3), nor to introduce Tycho's timing engine (Chapter 4). Rather, it establishes the empirical constraints imposed by the telemetry sources themselves. These include sensor refresh

intervals, stability of consecutive updates, delays between physical behaviour and reported values, the presence or absence of timestamps, and the distinction between instantaneous versus internally averaged measurements.

The subsections that follow describe these temporal properties for each telemetry source individually and summarise the practical limits they impose on high-resolution energy analysis.

2.3.1 RAPL Update Intervals and Sampling Stability

RAPL exposes energy *counters* rather than instantaneous power values. These counters accumulate energy since boot and can be read at arbitrarily high frequency, but their usefulness is determined entirely by how often the internal measurement logic refreshes them, a timing behaviour that is undocumented and domain-dependent.

Domain-specific internal update rates. Intel specifies the RAPL time unit as 0.976 ms for the slowest-updating domains, while others, notably the PP0 (core) domain, may refresh significantly faster [21]. In practice, however, these theoretical limits do not translate into usable temporal resolution because RAPL provides no timestamps: the moment of counter refresh is unknown to the reader. At sub-millisecond sampling rates, the lack of timestamps combined with irregular refresh behaviour introduces substantial relative error, since differences between consecutive reads may reflect counter staleness rather than actual power dynamics [23].

Noise introduced by security-driven filtering. To mitigate power-side channels such as Platypus, Intel optionally introduces randomised noise through the `ENERGY_FILTERING_ENABLE` mechanism [26]. This filtering increases the effective minimum granularity from roughly 1 ms to approximately 8 ms for the PP0 domain [21]. While average energy over longer intervals remains accurate, instantaneous increments become less reliable at very short timescales.

Practical sampling limits. Despite the nominal sub-millisecond timing, empirical work consistently shows that high-frequency polling offers no practical benefit. Multiple studies report that sampling faster than the internal update period only produces repeated counter values and amplifies read noise [23]. Jay et al. demonstrate that at polling rates slower than 50 Hz, the relative error falls below 0.5 % [24]. Consequently, typical measurement practice (and the limits adopted in this thesis) treats RAPL as reliable only at tens-of-milliseconds resolution, not at the theoretical millisecond scale suggested by its nominal time unit.

Summary. Although RAPL counters can be read extremely quickly, the effective temporal resolution is constrained by undocumented refresh intervals, absence of timestamps, optional security filtering, and substantial measurement noise at high polling rates. For practical purposes, sampling at approximately 20–50 ms intervals yields the most stable and accurate results, while sub-millisecond polling is inadvisable due to high relative error and counter staleness.

2.3.2 GPU Update Intervals and Sampling Freshness

GPU power telemetry is exposed primarily through NVML, with DCGM providing an alternative access path that builds on the same underlying measurement

source. Unlike CPU-side interfaces integrated into the processor package, GPU power monitoring is performed entirely by the device itself: internal sensing circuits and firmware determine how often new values are produced, how they are averaged, and when they are published to software. As a result, refresh behaviour varies substantially across architectures, and the temporal properties of the reported values depend on device-internal update cycles rather than the rate at which the host system issues queries, which limits the achievable resolution of any external sampling strategy.

Internal update cycles and sampling freshness. Empirical studies consistently show that NVML publishes new power values only intermittently, even when queried at high frequency. Yang et al. report sampling availability as low as roughly twenty–twenty-five percent across more than seventy modern data-center GPUs, meaning that the majority of polls return previously published values rather than fresh measurements [27].

Typical internal update periods fall on the order of tens to several hundreds of milliseconds, with architectural variation between GPU generations. Hernandez et al. report that newer architectures apply more aggressive smoothing and exhibit longer gaps between updates, reflecting slower publication cadence at the firmware level [28]. Overall, empirical evaluations show that NVML's internal update interval may lie on the order of hundreds of milliseconds and that repeated queries do not guarantee the retrieval of a new sample at every call [27]. NVML power readings do not represent instantaneous electrical measurements; they reflect firmware-level integration and smoothing over a device-internal averaging window, the duration of which varies by GPU generation and is not publicly documented..

Reaction delay to workload-induced power changes. A related characteristic is NVML's reaction delay: when GPU power changes due to workload activity, the corresponding update becomes visible only after a lag. Multiple studies document delays in the range of approximately one to three hundred milliseconds before a new NVML value reflects the underlying power transition [27]. This delay is distinct from averaging effects and arises from deferred publication of internally accumulated measurements. On some recent architectures, the delay can be longer due to device-level smoothing layers that defer updates until sufficient internal samples have been collected [28].

Update regularity and jitter. NVML update cycles are not perfectly periodic. Even when a nominal internal cadence is observable, individual publish times exhibit modest jitter, and occasional missed or skipped updates can result in sequences of identical values. These effects are pronounced on certain consumer-class devices and in configurations that partition the GPU, such as MIG, although they are also present to a lesser degree on data-center accelerators [27]. Such irregularity introduces uncertainty regarding the true measurement time of any retrieved value, especially in the sub-second range.

DCGM sampling behaviour. DCGM relies on the same underlying measurement path as NVML and therefore inherits NVML's internal update characteristics. In practice, DCGM is commonly accessed through its exporter, which introduces an additional periodic sampling stage (typically around one second) resulting in markedly

coarser temporal behaviour than NVML's native cadence. As a result, DCGM-based power telemetry rarely offers sub-second resolution in operational environments [29].

GPU utilization update cycles. NVML's GPU utilization metric follows its own internal update cadence, separate from power. It is typically refreshed more frequently (on the order of tens of milliseconds) although the exact timing remains undocumented. While this metric does not track computational efficiency, its shorter update interval provides a comparatively more responsive indicator of device activity [29].

2.3.3 Redfish Sensor Refresh Intervals and Irregularity

Redfish exposes power telemetry through the baseboard management controller (BMC) and therefore inherits the temporal behaviour of its embedded sensing hardware and firmware. In contrast to on-chip interfaces such as RAPL or NVML, Redfish is designed for management-plane observability rather than high-frequency monitoring. Prior studies consistently report that Redfish refreshes whole-node power values at coarse intervals, typically ranging from several hundred milliseconds to multiple seconds, with the exact cadence depending on vendor, BMC firmware, and underlying sensor design [4, 14]. The Redfish standard does not define a minimum update frequency, and available documentation provides little insight into internal sampling or averaging strategies.

Measurement semantics of Redfish power values. Redfish does not expose instantaneous electrical measurements. Instead, the reported values originate from on-board monitoring chips connected to shunt-based sensors and are subsequently processed inside the BMC. Vendor documentation indicates that these sensors inherently integrate power over tens to hundreds of milliseconds, and that additional firmware-level smoothing may be applied before values are published through the Redfish API [14]. Empirical evaluations support this interpretation: Wang et al. show that Redfish exhibits reaction delays of roughly two hundred milliseconds and displays particularly stable behaviour under steady loads, consistent with block-averaged rather than instantaneous sampling [4]. Because neither the sensor integration window nor any BMC filtering policies are defined in the standard, the temporal semantics of published values remain implementation-dependent.

Redfish power readings include a timestamp field, but this value reflects the BMC's observation time rather than the sampling instant of the physical power sensor. In many implementations, timestamps are rounded to seconds, which limits their utility for reconstructing sub-second dynamics and prevents reliable inference of the underlying sampling moment.

Beyond published work, empirical observations from the system used in this thesis reveal that Redfish update intervals may exhibit substantial variability. While nominal refresh periods appear regular over longer windows, individual samples occasionally show multi-second gaps, repeated values, or irregular spacing. Such behaviour is consistent with a telemetry source operating on management-plane scheduling and BMC workload constraints rather than real-time guarantees. These observations do not generalise across vendors but illustrate the degree of temporal uncertainty that can occur in practice.

Overall, Redfish provides a widely supported mechanism for obtaining whole-system power readings and is well suited for coarse-grained monitoring or validation of other telemetry sources. Its coarse refresh intervals, lack of sensor-level timestamps, and implementation-dependent irregularities, however, make it unsuitable for analysing short-duration phenomena or for use as a primary source in high-resolution energy attribution.

2.3.4 Timing of Software-Exposed Metrics

Software-exposed resource metrics differ fundamentally from hardware-integrated telemetry sources: rather than publishing sampled power or energy values at device-defined intervals, the Linux kernel exposes cumulative counters whose temporal behaviour is almost entirely determined by when they are read. These interfaces therefore provide quasi-continuous visibility into system activity, but without intrinsic update cycles or timestamps that would define the sampling moment of the underlying measurement.

Cumulative counters in `/proc` and cgroups. Kernel interfaces such as `/proc/stat`, per-task entries under `/proc/<pid>`, and the CPU accounting files in cgroups expose resource usage as monotonically increasing counters. These values are updated by the kernel during scheduler events, timer interrupts, and context-switch accounting, rather than at fixed intervals. As a consequence, their effective temporal resolution is determined entirely by the user's polling cadence: reading them more frequently produces more detailed deltas, but the kernel does not provide any guarantee about when a counter was last updated. None of these counters include timestamps, and their update timing may vary across systems due to tickless operation, kernel configuration, and workload characteristics.

Disk and network I/O statistics. I/O-related counters follow the same principle. Entries such as `/proc/<pid>/io`, cgroup I/O files, and interface statistics in `/proc/net/dev` are incremented as part of the corresponding driver paths when I/O operations occur. They do not refresh periodically and therefore exhibit update patterns that mirror workload activity rather than a regular cadence. Temporal interpretation again depends entirely on the polling rate of the monitoring system.

eBPF-based event timing. In contrast to cumulative counters, eBPF enables event-driven monitoring with explicit timestamps. Kernel probes attached to scheduler events, I/O paths, or tracepoints can record event times with high precision using the kernel's monotonic clock. As a result, eBPF metrics provide effectively instantaneous temporal resolution and are limited only by the overhead of probe execution and user-space consumption of BPF maps. No internal refresh cycle exists; events are timestamped at the moment they occur.

Performance counters and `perf`-based monitoring. Hardware performance monitoring counters (PMCs), accessed via `perf_event_open`, advance continuously within the processor. They do not follow a publish interval, and their timing semantics are defined solely by the instant at which user space reads the counter. This provides fine-grained and low-latency access to execution metrics such as cycles and retired instructions, with overhead rising only when polling is performed at very high frequencies.

Overall, software-exposed metrics behave as cumulative or event-driven signals rather than sampled telemetry sources. Their temporal characteristics are dominated by polling strategy and kernel-level event timing, with eBPF representing the only interface that attaches precise timestamps directly to system events.

2.4 Existing Tools and Related Work

Energy observability in containerized environments has attracted increasing attention in recent years, leading to the development of several tools that combine hardware, software, and statistical telemetry to estimate per-workload energy consumption. Despite this diversity, only a small number of tools attempt to attribute energy at container or pod granularity with sufficient detail to inform system-level research. Among these, *Kepler* has emerged as the most widely adopted open-source solution within the cloud-native ecosystem, while *Kubewatt* represents the first focused research effort to critically evaluate and refine Kepler's attribution methodology. Other frameworks, such as Scaphandre, SmartWatts, or PowerAPI, offer relevant ideas but differ in scope, telemetry assumptions, or operational goals. For this reason, the remainder of this section concentrates primarily on Kepler and Kubewatt, using these two tools to illustrate the architectural and methodological challenges that motivate the research gaps identified at the end of this chapter.

2.4.1 Kepler

2.4.1.1 Architecture and Metric Sources

Kepler[49] is a node-local energy observability agent designed for Kubernetes environments. Its architecture follows a modular dataflow pattern: a set of collectors periodically ingests telemetry from hardware and kernel interfaces, an internal aggregator aligns and normalizes these inputs, and a Prometheus exporter exposes the resulting metrics at container, pod, and node granularity. This structure allows Kepler to integrate heterogeneous telemetry sources while presenting a unified metric interface to external monitoring systems.

Kepler's collectors obtain process, container, and node telemetry from standard Linux and Kubernetes subsystems. Resource usage statistics are taken from `/proc`, cgroup hierarchies, and Kubernetes metadata, while hardware-level energy data is read from RAPL domains via the `powercap` interface. Optional collectors provide GPU metrics through NVIDIA's NVML library and platform-level power measurements via Redfish or other BMC interfaces. All inputs are treated as cumulative counters or periodically refreshed state, and their effective resolution is therefore determined by Kepler's sampling configuration. All metrics are updated at same interval and at the same time (default: 60 seconds for redfish, 3 seconds for all other sources). A central responsibility of the aggregator is to map raw per-process telemetry to containers and pods, using cgroup paths and Kubernetes API metadata. The derived metrics are finally exposed via a Prometheus endpoint, enabling integration into common cloud-native observability stacks.

In contrast to generic system monitoring agents, Kepler's architecture is tailored specifically to Kubernetes. Its emphasis on container metadata, cgroup-based accounting, and workload-oriented metric aggregation distinguishes it from tools that operate primarily at the host or VM level. At the same time, its reliance on standard

Linux interfaces keeps deployment overhead low, requiring only node-local access to `/proc`, cgroups, and the `powercap` subsystem.

Overall, Kepler's architectural design reflects a trade-off between flexibility and granularity: while it can ingest diverse telemetry sources and attribute energy at container level, its accuracy is constrained by the timing and resolution of the underlying metrics, as well as the unified sampling cadence chosen for the collectors.

Kepler updates all metrics within a single synchronous loop that triggers every sampling interval. This design simplifies integration but enforces a uniform cadence across heterogeneous telemetry sources, which contributes to the timing and alignment issues discussed in § 2.4.1.3. The structure is shown in Figures 2.1.

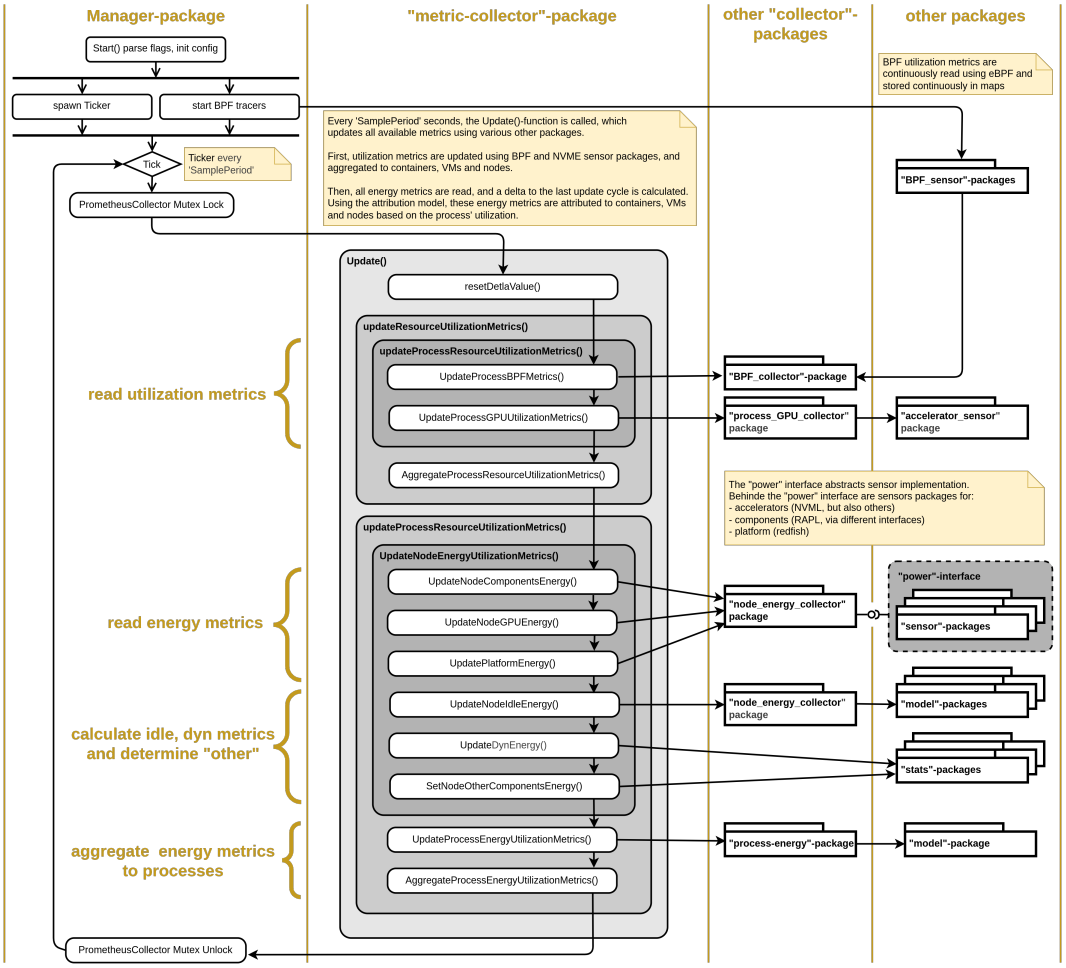


FIGURE 2.1: Kepler's synchronous update loop, where all collectors run at a unified sampling interval.

2.4.1.2 Attribution Model

Kepler's attribution logic follows a two-stage structure. First, node-level energy is decomposed into *idle* and *dynamic* components for each available power domain (package, core, uncore, DRAM, and optionally GPU or platform-level readings). Second, the dynamic portion is distributed across processes and containers according to their observed resource usage, while idle power is assigned using a domain-specific

default policy.

Dynamic power is attributed proportionally using ratio-based models. For each domain, Kepler computes the energy delta over the sampling interval and distributes it according to a usage metric selected for that domain. Instructions, cache misses, and CPU time are used as primary signals, with fallbacks when a metric is unavailable. GPU dynamic power is attributed based on GPU compute utilization. Platform-level power, when available, is treated as a residual domain: after subtracting CPU and DRAM power, the remaining portion is shared across workloads using the designated default metric or, if none is configured (which is the case), an equal split.

Idle energy is handled separately. Kepler maintains a rolling estimate of minimum node-level power for each domain and treats these values as idle baselines. In the default configuration, idle energy is divided evenly across all active workloads during the attribution interval. While this behaviour differs from protocol recommendations that scale idle power by container size, Kepler applies a uniform policy across domains to ensure attribution completeness.

Attribution operates at process granularity, with container and pod values obtained by aggregating the processes mapped to each cgroup. This approach allows Kepler to attribute energy to short-lived or multi-process containers while retaining compatibility with Kubernetes metadata.

Kepler performs attribution at a fixed internal update interval (default: 3 s). All usage metrics and energy deltas within that interval are aggregated before attribution is computed. Because Prometheus scrapes occur independently of Kepler's internal loop, the exported time series may reflect misalignment when the scrape period is not a multiple of the update interval. This can lead to visible step patterns or oscillations, particularly for dynamic workloads. Despite these limitations, the model provides a coherent and workload-oriented view of node-level energy consumption suitable for cloud-native observability scenarios.

2.4.1.3 Observed Behavior and Limitations

Several studies and code-level inspections reveal that Kepler's attribution behaviour exhibits systematic limitations that affect accuracy and interpretability. The most comprehensive empirical evaluation to date, conducted by Pijnacker et al., demonstrates that attribution inaccuracies arise even when node-level power estimation is reliable[50]. Their experiments highlight that idle power is often distributed to containers that are no longer active, including Completed pods, and that dynamic power can be reassigned inconsistently when containers are added or removed. These effects stem from the coupling of process-level accounting with container lifecycle events, which may lag behind cgroup or Kubernetes metadata updates.

Timing mismatches further contribute to attribution artifacts. High-frequency CPU and cgroup statistics are combined with slower telemetry sources such as Redfish, whose update intervals may span tens of seconds. When workload intensity changes during such periods, Kepler may assign disproportionately large or small dynamic energy shares to individual containers. Similar behavior occurs at the Prometheus interface when scrape intervals do not align with Kepler's internal update loop, producing visible oscillations in the exported time series.

Source-code inspection reinforces these observations. Numerous unimplemented or placeholder sections (e.g. TODO markers) affect key components of the ratio-based attribution model and default configuration paths. In particular, some domains lack explicit usage metrics, leading to fallback behaviour and equal-cost splitting regardless of container activity. GPU attribution relies on a single utilization metric and is therefore sensitive to the temporal behaviour of NVML's sampling. Together, these issues introduce variability across domains and reduce the transparency of the resulting per-container energy values.

Lifecycle handling also presents challenges. Because container metadata is aggregated from process-level information, short-lived or Completed pods may retain residual energy assignments. Conversely, system processes that cannot be mapped cleanly to Kubernetes abstractions may absorb unassigned power, obscuring the relationship between application behaviour and observed consumption.

Overall, these limitations underscore that Kepler provides a practical but imperfect approximation of container-level energy consumption. The observed behaviour motivates the research gaps identified at the end of this section, particularly the need for finer temporal resolution, explicit handling of idle and residual power, configurable attribution models, and more robust reconciliation across heterogeneous telemetry sources.

2.4.1.4 Kepler v0.10.0

In July 2025, Kepler underwent a substantial architectural redesign with the release of version 0.10.0[51]. The new implementation replaces many of the privileged operations used in earlier versions, removing the need for `CAP_BPF` or `CAP_SYSADMIN` and reducing reliance on kernel instrumentation. Instead, Kepler now obtains all workload statistics from read-only `/proc` and cgroup interfaces. This reduction in privilege requirements significantly improves deployability and security, particularly for managed Kubernetes environments where eBPF- or perf-based approaches are infeasible.

The redesign also introduces a markedly simplified attribution model. Whereas earlier versions combined multiple hardware and software counters (e.g. instructions, cache misses, GPU utilization) to estimate dynamic energy, Kepler v0.10.0 relies exclusively on CPU time as the usage metric. Node-level dynamic energy is computed by correlating RAPL deltas with aggregate CPU activity, and each workload receives a proportional share of this value based solely on its CPU time fraction. Idle energy is not distributed to containers and instead remains part of a node-level baseline. Containers, processes, and pods are treated as independent consumers drawing from the same active-energy pool, with no dependence on process-derived aggregation.

These changes increase robustness and predictability: the simplified model is easier to reason about, less sensitive to heterogeneous workloads or timing mismatches, and compatible with environments where kernel-level measurement facilities are unavailable. However, the loss of metric flexibility substantially reduces modeling fidelity. Fine-grained distinctions between compute-bound and memory-bound tasks are no longer observable, and the attribution model presumes a strictly linear relationship between CPU time and power consumption. As a result, Kepler v0.10.0 no longer targets high-accuracy energy attribution but instead emphasises operational stability and minimal overhead.

For the purposes of this thesis, Kepler v0.10.0 is relevant primarily as an indication of the project's strategic shift toward simplicity and broad deployability. Its CPU-time-only model is not suitable as a basis for Tycho, whose objectives require higher temporal resolution, more diverse metric inputs, and explicit handling of domain-level energy contributions. Accordingly, the remainder of this chapter focuses on the behaviour of Kepler v0.9.x, which remains the most representative version for research-oriented attribution discussions.

2.4.2 KubeWatt

KubeWatt is a proof-of-concept exporter developed by Pijnacker as a direct response to the attribution issues uncovered in Kepler.[50, 52] Rather than extending Kepler's complex pipeline, KubeWatt implements a deliberately narrow but transparent model that focuses on correcting three specific problems: misattribution of idle power, leakage of energy into generic "system processes", and unstable behaviour under pod churn. It targets Kubernetes clusters running on dedicated servers and assumes that a single external power source per node is available (in the prototype, Redfish/iDRAC).

A central design decision is the strict separation between *static* and *dynamic* power. Static power is defined as the baseline cost of running the node and its control plane in an otherwise idle state. KubeWatt measures or estimates this baseline once and treats it as a constant; it is *not* attributed to containers. Dynamic power is then computed as the difference between total node power and this static baseline and is the only quantity distributed across workloads. Control-plane pods are explicitly excluded from the dynamic attribution set, so their idle consumption remains part of the static term and does not pollute application-level metrics.

To obtain the static baseline, KubeWatt provides two initialization modes. In *base initialization*, the cluster is reduced to an almost idle state (only control-plane components), and node power is sampled for a few minutes. The static power value is computed as a simple average, yielding a highly stable estimate under the test conditions. When workloads cannot be stopped, *bootstrap initialization* fits a regression model to time series of node power and CPU utilization collected during normal operation. The regression is evaluated at the average control-plane CPU usage to infer the static baseline. This mode is more sensitive to workload characteristics and SMT effects but provides a practical fallback when base initialization is not feasible.

During normal operation, KubeWatt runs in an *estimation mode* that attributes dynamic node power to containers proportionally to their CPU usage. CPU usage is obtained from the Kubernetes metrics API (`metrics.k8s.io`) at node and container level. The denominator explicitly sums only container CPU usage; system processes, cgroup slices, and other non-container activity are excluded by construction. This corrects a key source of error in Kepler, where slice-level metrics and kernel processes could receive non-trivial fractions of node power. Under stable workloads and at the relatively coarse sampling interval used in the prototype, KubeWatt achieves container-level power curves that align well with both iDRAC readings and observed CPU utilisation, and it behaves robustly when large numbers of idle pods are created and deleted.

The scope of KubeWatt is intentionally narrow. It is CPU-only, uses a single external power source per node, assumes that Kubernetes is the only significant workload

on the machine, and does not attempt to model GPU, memory, storage, or network energy. It also inherits the temporal limitations of the Kubernetes metrics pipeline and treats Redfish power readings as instantaneous, without explicit latency compensation. Nevertheless, KubeWatt demonstrates that a simple, well-documented ratio model with explicit static–dynamic separation and strict cgroup filtering can eliminate several of Kepler's most problematic attribution artefacts. These design principles are directly relevant for the attribution redesign pursued in this thesis and inform the requirements placed on Tycho's more general, multi-source architecture.

2.4.3 Other Tools (Brief Overview)

Beyond Kepler, several tools illustrate the methodological diversity in container- and process-level energy attribution, although they are not central to the Kubernetes-specific challenges addressed in this thesis. Scaphandre[53] provides a lightweight proportional attribution model based exclusively on CPU time and RAPL deltas. Its design emphasises simplicity and portability, offering basic container mapping through cgroups but limited control over sampling behaviour or attribution semantics. SmartWatts[54], by contrast, represents a more sophisticated approach: it builds performance-counter-based models that self-calibrate against RAPL measurements and adapt dynamically to the host system. While effective in controlled environments, SmartWatts requires access to perf events, provides only CPU and DRAM models, and is not deeply integrated with Kubernetes abstractions.

A broader ecosystem of lightweight tools (e.g. CodeCarbon[55] and related library-level estimators) demonstrates further variation in scope and assumptions, but these generally target high-level application profiling rather than system-wide workload attribution. Collectively, these tools highlight a spectrum of design choices (from simplicity and portability to model-driven estimation) but none address the combination of high-resolution telemetry, multi-tenant attribution, and Kubernetes metadata integration that motivates the development of Tycho.

2.4.4 Cross-Tool Limitations Informing Research Gaps

Across the surveyed tools, several structural limitations recur despite substantial differences in design philosophy and implementation. First, temporal granularity remains insufficient: although hardware interfaces such as RAPL support millisecond-level updates, most tools aggregate measurements over multi-second intervals. This obscures short-lived workload behaviour and reduces attribution fidelity, particularly in heterogeneous or bursty environments. Second, all tools depend on telemetry sources whose internal semantics are only partially documented. Ambiguities regarding RAPL domain coverage, NVML power reporting, or BMC-derived node power constrain both the interpretability and the auditability of reported metrics, reinforcing the black-box character of current measurement pipelines.

Idle-power handling presents a further source of inconsistency. Tools differ widely in how idle power is defined, whether it is attributed, and to whom. These choices are often implicit, undocumented, or constrained by implementation artefacts, leading to attribution patterns that are difficult to interpret or reproduce. Multi-domain coverage is similarly limited: existing tools focus primarily on CPU and, to a lesser extent, DRAM or GPU consumption, leaving storage, networking, and other subsystems unmodelled despite their relevance to node-level energy use.

Metadata lifecycle management also emerges as a common limitation. Rapid container churn, transient pods, and the interaction between Kubernetes and cgroup identifiers can produce incomplete or stale workload associations, affecting attribution stability. Finally, attribution models themselves are typically rigid. Most tools hard-code a specific proportionality assumption (commonly CPU time or a single hardware counter) and provide limited support for calibration, uncertainty quantification, or alternative modelling philosophies.

Taken together, these limitations reveal structural gaps in current approaches to container-level energy attribution. They motivate the need for tools that combine high-resolution telemetry handling, transparent and configurable attribution logic, robust metadata management, and principled treatment of uncertainty. The next section distills these observations into concrete research gaps that inform the design objectives of Tycho.

2.5 Research Gaps

This section synthesises the findings from the preceding analyses of telemetry sources, temporal behaviour, and existing tools. Across these perspectives, a set of structural limitations emerges that fundamentally constrains accurate and explainable energy attribution in Kubernetes environments. These limitations arise at three intertwined layers: the measurement interfaces exposed by hardware and kernel subsystems, the attribution models built on top of these measurements, and the operational context in which Kubernetes workloads execute. Taken together, they demonstrate the absence of a framework that provides high temporal precision, transparent modelling assumptions, and robustness to container lifecycle dynamics. The gaps identified below define the technical requirements that motivate the design of Tycho.

(1) Measurement Gaps: Temporal Resolution and Telemetry Semantics

Existing tools do not exploit the full temporal capabilities of modern hardware telemetry. Interfaces such as RAPL offer fast, reliable update frequencies, yet tools operate on fixed multi-second loops, causing short-lived or bursty activity to be temporally averaged away. Moreover, latency mismatches between high-frequency utilization signals (e.g. cgroups, perf counters) and low-frequency power interfaces (e.g. Redfish/BMC) introduce structural attribution errors that are not explicitly modelled or corrected.

A related issue is the opacity of hardware telemetry. RAPL, NVML, and BMC power sensors provide indispensable data, but their domain boundaries, averaging windows, and internal update behaviour are insufficiently documented. This prevents rigorous interpretation of reported values and inhibits the development of calibration or uncertainty models. Finally, measurement coverage remains incomplete: while CPU and DRAM domains are widely supported, no standardised telemetry exists for storage, networking, or other subsystems. Current tools treat these components either implicitly (as part of “platform” power) or not at all.

(2) Attribution Model Gaps: Rigidity, Idle Power, and Domain Consistency

Current attribution models rely on rigid proportionality assumptions (typically CPU time, instructions, or a single hardware counter) without considering alternative modelling philosophies. Idle power remains a persistent source of inconsistency: tools variously divide it evenly, proportionally, or not at all, often without documenting the rationale. These choices have substantial effects on per-container energy values, particularly in lightly loaded or heterogeneous systems.

At the domain level, attribution methods are not unified. CPU, DRAM, uncore, GPU, and platform energy are treated through incompatible heuristics, and many domains fall back to equal distribution when no clear usage signal is defined. None of the surveyed tools quantify uncertainty, despite relying on noisy, coarse, or undocumented telemetry sources. As a result, attribution outputs appear deterministic even when they rest on incomplete or ambiguous measurement assumptions.

(3) Metadata and Lifecycle Gaps: Churn, Timing, and Virtualization

Container-level attribution requires consistent mapping between processes, cgroups, and Kubernetes metadata. Existing tools struggle in scenarios with rapid container churn, ephemeral or Completed pods, and multi-process containers. Mismatches between metadata refresh cycles and metric sampling lead to stale or missing associations, which propagate into attribution artefacts.

Energy attribution inside virtual machines remains essentially unsolved. No standard mechanism exists for exposing host-side telemetry to guest systems in a way that preserves temporal alignment and attribution consistency. The limited QEMU-based passthrough available in Scaphandre is not generalisable, and conceptual proposals (e.g. Kepler's hypercall mechanism) remain unimplemented. Given the prevalence of cloud-hosted Kubernetes clusters, this constitutes a major practical limitation.

(4) Usability, Transparency, and Operational Gaps

For most tools, implementation assumptions, fallback paths, and attribution decisions are implicit. Users cannot easily distinguish measured values from estimated ones, nor identify the assumptions underlying attribution outputs. This lack of transparency reduces trust and complicates debugging.

Operational constraints further restrict applicability. Tools that require privileged kernel instrumentation (eBPF, `perf_event_open`) are unsuitable for many production clusters, while tools designed around unprivileged access often sacrifice modelling fidelity. At the same time, developers, operators, and researchers have fundamentally different observability needs, yet existing tools optimise for only one audience at a time. None provide configurable attribution modes or role-specific abstractions.

(5) Missing Support for Calibration and Validation

Beyond isolated exceptions, existing tools provide limited mechanisms for systematic calibration or validation of their attribution models. KubeWatt is one of the few

systems that performs explicit baseline calibration, offering both an idle-power measurement mode and a statistical fallback for environments without idle windows. Kepler offers no structured calibration workflow, and its estimator models lack reproducible training procedures. SmartWatts introduces online model recalibration but focuses narrowly on performance-counter regression, leaving node-level baselines, multi-domain alignment, and external ground-truth integration unaddressed.

Across all tools, there is no standardized path to incorporate external measurements (for example from wall-power sensors or BMC-level telemetry) to validate or refine model behaviour. Idle power is seldom isolated as a first-class parameter, attribution error is rarely quantified, and no system provides uncertainty estimates that reflect measurement or modelling limitations. Without such calibration and validation capabilities, attribution accuracy cannot be assessed, corrected, or improved over time—an essential requirement for any system intended to provide trustworthy, high-resolution energy insights in Kubernetes environments.

2.6 Summary

The analyses in this chapter reveal that modern server platforms provide a heterogeneous and only partially documented set of telemetry interfaces whose temporal and semantic properties fundamentally constrain container-level energy attribution. Hardware-integrated sources such as RAPL and NVML expose valuable domain-level energy information but differ substantially in update behaviour, averaging semantics, and domain completeness. Out-of-band telemetry via Redfish provides stable whole-system measurements but at coarse and irregular temporal granularity. Software-exposed metrics offer fine-grained visibility into workload behaviour, yet they measure utilisation rather than power and depend entirely on polling strategies for temporal interpretation.

Temporal irregularities, internal averaging, and undocumented sensor behaviour reduce the effective precision of all telemetry sources, especially when attempting to capture short-lived workload dynamics. Existing tools aggregate these heterogeneous signals using fixed multi-second sampling loops and rigid proportionality assumptions, which leads to systematic attribution artefacts. Idle power is treated inconsistently across systems, residual power is frequently conflated with workload activity, and multi-domain attribution remains fragmented in both semantics and implementation. Metadata churn and asynchronous refresh cycles further complicate the mapping between processes, containers, and Kubernetes abstractions, reducing the stability and interpretability of attribution outputs.

The cross-tool evaluation confirms these structural limitations. Kepler provides broad telemetry integration but struggles with timing mismatches, incomplete domain semantics, and opaque idle handling. Kubewatt demonstrates the importance of explicit baseline separation and cgroup filtering, yet remains limited to single-domain CPU-based estimation. Other tools illustrate a spectrum of design choices but do not address the combined challenges of high-frequency telemetry, multi-domain attribution, container lifecycle dynamics, and Kubernetes integration.

Together, these findings motivate the need for a framework that provides high temporal fidelity, transparent modelling assumptions, unified domain treatment, explicit handling of idle and residual energy, and robust metadata reconciliation. These

requirements form the conceptual and architectural foundations developed in Chapter 3, which introduces the methodological principles guiding the design of Tycho.

Chapter 3

Conceptual Foundations of Container-Level Power Attribution

Energy attribution explains how workloads contribute to the power consumed by a node. Hardware exposes only aggregate energy behaviour, so attribution constructs a model that distributes this aggregate across multiple sources of activity. The aim of this chapter is to establish the conceptual basis for such modelling. It introduces the abstractions needed to reason about workloads, execution units, temporal structure, and observation windows. It also presents the principles that constrain any defensible attribution model and identifies the interactions that make attribution non-trivial. The discussion is purely conceptual and does not rely on empirical detail from Chapter 2. These concepts form the foundation for the system requirements developed later in the chapter and for the architectural design presented in Chapter 4.

3.1 Nature and Purpose of Power Attribution

Power attribution is a modelling activity. Hardware reports only aggregate power or energy, and attribution constructs an explanation that distributes this aggregate across the workloads running on the node. The model is necessarily reactive because measurements become available only after the underlying activity has occurred. Attribution therefore explains past energy behaviour rather than providing realtime insight, although the resulting information can support optimisation and accountability.

Attribution is useful because it reveals how different workloads contribute to dynamic power consumption. This enables comparisons between workloads, supports evaluation of deployment or scheduling strategies, and provides interpretable information for higher level policy decisions.

For the purposes of this chapter, a workload is defined as a logical entity that groups one or more execution units into a stable attribution target. Execution units may include processes, threads, containers, virtual machines, or service components. Their membership can change over time, but the workload identity persists as the object to which energy is assigned.

3.2 Workload Identity and Execution Boundaries

Energy attribution operates on workload identities rather than on individual execution units. Execution units such as processes, threads, or container instances appear and terminate independently, often with lifetimes that do not align with observation windows. Their behaviour may overlap, interleave, or succeed one another in ways that complicate any direct mapping between activity and energy. A workload need not coincide with a single application; it may represent a subset of an application, a combination of cooperating services, or a logical grouping chosen purely for attribution.

Attribution therefore relies on a stable abstraction that groups such units into coherent entities. Orchestration frameworks, including systems such as Kubernetes, illustrate this principle by associating container instances with higher level constructs such as pods or services. The attribution target is the logical workload, not the transient units that realise it at any moment.

Short lived execution units raise specific challenges. Some may terminate between consecutive measurements, and their activity may be only partially observable. Others may overlap in time while belonging to the same workload identity. A consistent attribution model must track these changing memberships without losing energy when units disappear or double counting energy when multiple units contribute concurrently. Stable workload identities provide the conceptual basis for such tracking.

3.3 Principles of Workload-Level Energy Attribution

Several principles constrain how node-level energy can be attributed to workloads. These principles are intertwined and reflect structural properties of shared hardware, the limits of observability, and the semantics of available metrics. Some depend on how measurements are structured in time, while others remain independent of temporal detail. The temporal aspects are developed further in § 3.4, but the principles themselves abstract from any specific system and define the conditions that any defensible attribution model must satisfy.

3.3.1 Aggregated Hardware Activity

Hardware exposes only aggregate power or energy for the node or for coarse hardware domains. It does not reveal how much of this consumption originates from any specific workload. Attribution therefore begins with a single observable quantity that reflects the combined activity of all execution units. Any per workload assignment is an inferred decomposition of this aggregate and must remain consistent with the measured total.

3.3.2 Domain Decomposition

Total system power is composed of contributions from several hardware domains, such as compute, memory, accelerators, storage, and platform circuitry. These domains respond differently to workload behaviour, and their relative impact varies across systems. Attribution must therefore reason at the domain level before assigning energy to workloads. Without such decomposition, the resulting assignments

would combine unrelated forms of activity and obscure the link between workload characteristics and observed power.

3.3.3 Conservation

Node-level energy is a fixed quantity within any observation window. An attribution model must assign energy to workloads in a way that is consistent with this total. The sum of all assigned energy, including any explicitly modelled background components, must equal the measured dynamic energy. Violations of conservation indicate that the model is incomplete or internally inconsistent.

3.3.4 Static–Dynamic Separation

System power consists of a baseline component that persists regardless of workload activity and a dynamic component induced by the workloads. Attribution concerns only the dynamic portion, so the baseline must be treated explicitly rather than absorbed into workload assignments. Any remaining unexplained energy must appear as a residual component and must not be redistributed silently across workloads.

3.3.5 Uncertainty and Non-Uniqueness

Workload-level energy attribution has no unique ground truth. Limited observability, asynchronous measurements, and interactions between hardware domains allow multiple decompositions of the same aggregate energy to be consistent with the measurements. A defensible attribution model must acknowledge this non-uniqueness and avoid implying precision that the underlying information does not support.

3.3.6 Dependence on Metric Fidelity

Attribution quality depends on the fidelity of the metrics that describe workload activity. Each metric has specific semantics, precision, and temporal resolution, and these properties determine how reliably the metric reflects the underlying hardware behaviour. An attribution model must therefore interpret metrics consistently and acknowledge that limited or coarse measurements constrain the accuracy of any inferred energy assignments.

Hardware subsystems are shared and not fully partitionable. Execution units contend for caches, memory controllers, and shared frequency or power budgets, and these interactions alter the relation between observed activity and actual power consumption. Such interference reduces the ability of any metric to isolate per workload effects and increases attribution uncertainty. A defensible model must incorporate these limitations when relating activity signals to domain level energy.

3.4 Temporal and Measurement Foundations

Attribution depends not only on which quantities are measured but also on when they are measured. Telemetry sources observe system behaviour at different times, with different implicit meanings, and with no inherent coordination. A clear temporal framework is therefore required to interpret workload activity and relate it to the energy observed at the node.

Observation Windows

Attribution operates on observation windows. A window integrates power and activity over a chosen duration and provides the temporal unit within which energy is assigned to workloads. All attribution reasoning occurs within these windows, so their boundaries determine which activity contributes to the measured energy and how temporal ambiguity affects attribution accuracy.

3.4.1 Sampling vs Event-Time Perspectives

Sampling records system state at fixed intervals, independent of when the underlying activity changes. Event time reflects the moment when the activity occurs or when a telemetry source updates its value. These perspectives rarely coincide. If a workload is active between two samples, the sampled values do not reveal when within the interval the activity occurred. Misalignment between when work happens and when it is observed creates ambiguity about how activity should be mapped into the observation window.

3.4.2 Clock Models and Temporal Ordering

Attribution requires a consistent ordering of events and measurements. Realtime clocks track wall clock time but may jump when synchronised, which breaks temporal ordering. Monotonic clocks advance continuously and therefore provide a stable basis for placing events on a time axis. A coherent attribution model relies on such ordering to determine which activity belongs to which observation window and to avoid artefacts caused by clock adjustments.

3.4.3 Heterogeneous Metric Sources

Telemetry originates from sources with different update cycles and semantics. Hardware counters accumulate events continuously and reveal activity only when read. Operating system accounting updates periodically according to scheduler behaviour. Device telemetry and external power interfaces publish measurements based on internal schedules. These sources do not share cadence, precision, or timestamp meaning. Their values represent different kinds of temporal information, and none can be assumed to align with the others.

3.4.4 Delay, Jitter, and Temporal Uncertainty

Measurements do not appear at the moment the underlying behaviour occurs. Observation delay arises when a metric is read after the activity has taken place. Publication delay arises when a telemetry source exposes an updated value only after internal processing. Jitter denotes variations in these delays. Because different sources exhibit different forms of delay, the temporal relation between activity and observed energy is uncertain. This uncertainty limits the precision with which activity can be linked to energy within an observation window.

3.4.5 Temporal Alignment of Asynchronous Signals

Attribution requires heterogeneous signals to be interpreted within the same observation window even though they arrive at different times and represent different temporal semantics. Some values describe cumulative changes, others instantaneous states, and others discrete events. A temporal alignment model must reconcile these signals without assuming true synchronisation. The goal is not to remove temporal uncertainty but to structure it so that attribution remains coherent and consistent with the measured energy.

3.5 Conceptual Attribution Frameworks

Because hardware exposes only aggregate energy, several modelling philosophies can be used to distribute this energy across workloads. These frameworks differ in how they relate activity metrics to energy and in how they treat uncertainty. None yields a unique solution, since the same measurements can support multiple plausible decompositions. Instead, each framework reflects a particular set of priorities, such as stability, fairness, or explanatory power, and provides a structured interpretation of the same underlying observations.

3.5.1 Proportional Attribution

Proportional attribution assigns energy to workloads in proportion to an observed activity metric, such as CPU time or memory access volume. Its appeal lies in its simplicity and interpretability. However, different metrics emphasise different forms of behaviour, and proportionality with respect to one metric does not imply proportionality with respect to another. The choice of metric therefore has direct consequences for the resulting attribution.

3.5.2 Shared-Cost Attribution

Shared-cost attribution distributes some portion of the dynamic energy uniformly or proportionally across all active workloads, independent of their individual activity levels. This approach emphasises stability and fairness and is often used when activity metrics are incomplete or unreliable. Its limitation is that it may obscure relationships between workload behaviour and energy consumption, since unexplained costs are not tied to specific activity.

3.5.3 Residual and Unattributed Energy

Some energy cannot be explained by available metrics or by direct workload activity. Subsystems without meaningful utilisation signals, background services, and asynchronous events contribute to a residual component. Treating this component explicitly preserves conservation and avoids distorting the energy assigned to observable activity. Residual energy also delineates the boundary between explainable and unexplained behaviour within an attribution model.

3.5.4 Model-Based or Hybrid Attribution

Model-based or hybrid attribution combines several activity signals into an explicit model of energy consumption. Such a model may weight metrics from different domains, encode domain-specific relationships, or blend proportional and shared-cost

components. It does not attempt to establish strict causality, but it treats the mapping from activity to energy as a structured function rather than a single proportional rule. The quality of the resulting attribution depends on how well the model captures the relevant relationships and on how stable these relationships remain across workloads and system states.

3.5.5 Causal or Explanatory Attribution

Causal or explanatory attribution attempts to relate changes in workload activity to changes in power consumption. It seeks to model relationships between metrics and energy rather than applying proportionality directly. This approach can capture more nuanced behaviour, but its accuracy depends on metric fidelity and on the stability of the relationship between activity and power. Limited observability and shared subsystem interactions restrict the strength of causal inferences.

Link to System Requirements

These frameworks illustrate the range of assumptions an attribution model may adopt. They highlight the need for transparent modelling choices, consistent interpretation of metrics, explicit treatment of residual components, and temporal coherence when relating activity to energy. In practice, their behaviour is further constrained by shared hardware, domain interactions, and temporal misalignment, which shape how any chosen framework behaves under real workloads. These combined effects are examined in § 3.6 and motivate the system requirements developed in § 3.7.

3.6 Interactions and Complications

The principles and temporal concepts introduced above interact in ways that make workload-level attribution fundamentally approximate. These interactions arise from shared hardware, limited observability, asynchronous measurements, and the structure of workloads themselves.

Combined Effects of Shared Hardware and Temporal Misalignment

Shared subsystems create interference that couples the activity of different workloads. Contention for caches, memory controllers, or shared power and frequency budgets alters the relation between observed activity and actual energy consumption. Temporal misalignment compounds this effect. When activity and power are observed at different times and with different delays, the ambiguity introduced by interference cannot be resolved by sampling alone. The combined effect limits the extent to which per workload contributions can be isolated.

Cross-Domain Interactions

Hardware domains are not independent. Changes in compute activity can influence memory behaviour or power states, and accelerators may shift platform level consumption. These interactions mean that energy attributed to one domain may reflect behaviour originating in another. Attribution must therefore operate under the constraint that domain boundaries provide structure but not complete separation.

Attribution as an Inverse Problem

Because only aggregate energy is measured, attribution requires inferring per workload contributions from incomplete and asynchronous observations. This inference is an inverse problem with multiple admissible solutions. Limited metric fidelity, shared hardware behaviour, and temporal uncertainty restrict how precisely activity can be mapped to energy. A coherent attribution model acknowledges these limitations and structures them explicitly rather than treating them as noise.

3.7 Conceptual Challenges and System Requirements

The challenges identified above arise from shared hardware behaviour, asynchronous and heterogeneous measurements, limited metric fidelity, and volatile workload lifecycles. Any attribution system must address these challenges within a coherent conceptual framework. The requirements formulated in this section follow directly from the principles and temporal foundations established earlier and specify the conditions that an attribution model shall satisfy.

3.7.1 Requirement: Temporal Coherence

The system *must* maintain coherent temporal structure across all telemetry sources. Measurements that arrive with differing delays, cadences, or timestamp semantics *shall* be placed on a consistent time axis and related correctly to the boundaries of the observation window. The system *should* tolerate irregular update patterns without introducing artefacts, and it *may* employ temporal reconstruction provided that ordering and conservation are preserved.

3.7.2 Requirement: Domain-Level Consistency

The system *must* decompose node-level energy into meaningful hardware domains before workload-level assignment. Each domain *shall* be treated using internally consistent rules, and the system *must not* combine unrelated forms of activity into a single attribution pathway. When direct observability is incomplete, the system *should* incorporate explicit residual modelling, and it *may* use domain specific strategies when justified by domain characteristics.

3.7.3 Requirement: Cross-Domain Reconciliation

The system *must* reconcile energy information from different hardware domains in a coherent manner. When domain-level signals disagree, the reconciliation strategy *shall* be explicit and internally consistent rather than relying on implicit priority rules. The system *should* expose when domains provide conflicting indications about energy usage and clarify how such conflicts influence per workload assignments. Any reconciliation *must not* violate conservation across domains or undermine the stability of workload-level attribution.

3.7.4 Requirement: Consistent Metric Interpretation

The system *must* interpret activity metrics in a stable and coherent manner. Metrics that differ in semantics, resolution, or precision *shall* not be combined without clear conceptual justification. The system *must not* allow the meaning of a metric to

vary across time or domains. It *should* treat metric limitations explicitly, and it *may* disregard metrics whose quality does not support meaningful attribution.

3.7.5 Requirement: Transparent Modelling Assumptions

All assumptions used to relate activity to energy *must* be explicit. The basis on which energy is distributed *shall* be interpretable, including the choice of attribution framework, the handling of idle and residual energy, and any fallback behaviour in the presence of incomplete metrics. The system *should* separate measured quantities from inferred quantities to avoid ambiguity, and it *may* expose configurable modelling options provided they do not violate consistency or conservation.

3.7.6 Requirement: Lifecycle-Robust Attribution

The system *must* remain consistent under workload churn. Execution units that appear or terminate within an observation window *shall* be tracked in a way that avoids both loss of energy and double counting. Workload identities *must* remain stable even when their underlying execution units change. The system *should* support overlapping lifecycles and transient units without degrading attribution quality, and it *may* use buffering or reconciliation strategies when necessary.

3.7.7 Requirement: Uncertainty-Aware Attribution

The system *should* acknowledge uncertainty arising from limited observability, shared hardware behaviour, and temporal misalignment. It *shall* avoid implying precision that the measurements do not support. Where feasible, it *should* represent unexplained energy explicitly rather than absorbing it into unrelated workloads. Any handling of uncertainty *must not* violate conservation or temporal ordering.

Link to Architectural Considerations

These requirements imply that an attribution system must provide mechanisms for temporal alignment, domain level reasoning, stable metric interpretation, explicit residual handling, and robust tracking of workload identities. They form the basis for the architectural design presented in Chapter 4.

3.8 Summary

This chapter introduced the conceptual foundations of workload-level energy attribution. It defined workloads and execution units, presented the principles that govern how aggregate energy can be decomposed, and developed the temporal and measurement concepts required to interpret heterogeneous telemetry. It also showed how shared hardware behaviour, metric limitations, and asynchronous observations interact to make attribution inherently approximate. These considerations led to a set of system requirements that any attribution model must satisfy. The next chapter builds on these requirements and introduces an architecture designed to meet them.

Chapter 4

System Architecture

4.1 Guiding Principles

Tycho's architecture is shaped by a small set of foundational principles that govern how measurements are interpreted, combined and ultimately attributed. These principles are architectural in nature: they articulate *how* the system must reason about observations, not *how* it is implemented. They establish the conceptual baseline that the subsequent sections refine in detail.

- **Accuracy-first temporal coherence.** Architectural decisions prioritise the reconstruction of temporally coherent views of system behaviour. Observations are treated as samples of an underlying physical process, and the architecture is designed to preserve their temporal meaning rather than force periodic alignment.
- **Domain-aware interpretation.** Metric sources differ in semantics and cadence. The architecture respects these differences and avoids imposing artificial synchrony or uniform sampling behaviour across heterogeneous domains.
- **Transparency of assumptions.** All modelling assumptions must be explicit, inspectable and externally visible. The architecture prohibits implicit corrections or hidden inference steps that would obscure how measurements lead to attribution results.
- **Uncertainty as a first-class concept.** Missing, stale or delayed information is treated as uncertainty rather than error. Architectural components convey and preserve uncertainty so that later stages may interpret it correctly.
- **Separation of observation, timing and attribution.** Measurement collection, temporal interpretation and energy attribution form distinct architectural layers. This separation prevents cross-coupling, clarifies responsibilities and ensures that improvements in one layer do not implicitly alter the behaviour of others.

4.2 Traceability to Requirements

The architectural structure introduced in this chapter provides a direct response to the requirements established in § 3.7. Each requirement class corresponds to specific architectural mechanisms, ensuring that the system design follows from formal constraints rather than implementation convenience.

Requirement: Temporal Coherence. Satisfied through event-time reconstruction, independent collector timelines, and window-based temporal alignment.

Requirement: Domain-Level Consistency. Addressed by per-domain interpretation layers, domain-aware handling of metric semantics, and explicit decomposition of node-level signals.

Requirement: Cross-Domain Reconciliation. Supported by a unified temporal model, window-level aggregation boundaries, and explicit reconciliation logic across domains during analysis.

Requirement: Consistent Metric Interpretation. Ensured by separating observation from interpretation, enforcing stable metric semantics within each domain, and isolating heterogeneous metrics into dedicated processing paths.

Requirement: Transparent Modelling Assumptions. Realised through explicit modelling steps, external visibility of assumptions, and separation between measured and inferred quantities.

Requirement: Lifecycle-Robust Attribution. Enabled by metadata freshness guarantees, stable process–container mapping, and attribution rules that remain valid under workload churn.

Requirement: Uncertainty-Aware Attribution. Supported by explicit treatment of stale or missing data, uncertainty propagation in window evaluation, and preservation of unexplained residuals.

4.3 High-Level Architecture

4.3.1 Subsystem Overview

Tycho is organised into a small set of subsystems, each with a distinct responsibility. The following overview introduces these subsystems without yet describing their interactions.

Timing engine. Defines the temporal reference used throughout the system and provides the notion of analysis windows. It is responsible for deciding when a window is complete and ready to be evaluated.

Metric collectors. Acquire observations from hardware and software sources and attach timestamps in the global temporal reference. They expose their output as streams of samples without coordinating with each other.

Metadata subsystem. Maintains the mapping between operating-system level entities and workload identities. It tracks relationships between processes, cgroups, containers and pods over time.

Buffering and storage layer. Stores recent observations in bounded histories so that samples relevant to a given window can be retrieved efficiently. It treats metric streams and metadata as read-mostly records.

Analysis engine. Interprets temporally aligned observations and metadata to produce energy estimates for each analysis window. It forms the logical bridge between

measurement and attribution.

Calibration framework. Derives auxiliary information about typical delays, update patterns and idle behaviour. It produces constraints and characterisations that other subsystems rely on for interpretation.

Exporter. Exposes the results of the analysis engine to external monitoring systems as metrics ready for scraping and downstream processing.

4.3.2 Dataflow and Control Flow

Before Tycho enters normal operation, external calibration scripts determine approximate delay characteristics for all relevant metric sources. At startup, Tycho's internal calibration component derives suitable polling frequencies for metric collectors and metadata acquisition, providing the initial operating parameters for the system.

During runtime, control flow originates in the timing engine. It triggers each collector according to its calibrated polling frequency, but collectors operate independently: they sample their respective domains without synchronising with each other, and each observation is timestamped and appended to a buffering layer together with its associated quality indicators. This buffering layer retains a bounded history of raw observations per metric domain, with a default retention duration of approximately 90 s. The retained history deliberately exceeds the duration of a single analysis window, providing extended temporal context for downstream analysis and modeling rather than limiting interpretation to window-local samples.

In parallel, metadata acquisition proceeds on its own schedule, refreshing the mappings between processes, cgroups and workload identities in the metadata cache. Metadata is not synchronised with metric collection but is interpreted jointly with buffered observations during analysis.

The timing engine also governs when analysis occurs. At regular intervals, constituting fixed-length analysis windows, it initiates a new evaluation cycle irrespective of how many samples have been collected within the most recent window. Each cycle begins by estimating idle behaviour for the relevant hardware domains based on the buffered observation history. The analysis engine then interprets buffered metric samples, metadata and idle characterisations, taking calibrated delay characteristics into account when reconstructing the temporal structure of the window. Although attribution is performed only for the current window, historical observations within the retention horizon inform delay interpretation, baseline estimation and other modeling steps.

Once analysis completes, the exporter publishes the resulting metrics in a form suitable for ingestion by external monitoring systems. Calibration remains active in the background throughout the system's lifetime: it observes collector behaviour and derived quantities over longer time spans and refines its characterisations when needed, informing both the timing and analysis components without altering any collected data.

Figure 4.1 provides a consolidated view of Tycho's control flow and data flow, highlighting the buffering layer as the bounded temporal substrate that decouples collection, analysis and export.

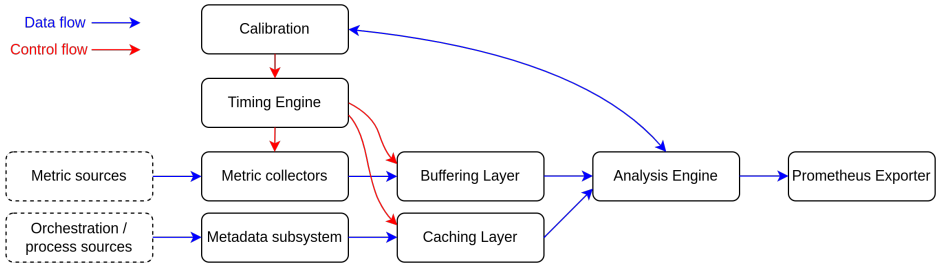


FIGURE 4.1: Subsystem Architecure, Dataflow and Control Flow

4.4 Temporal Model and Timing Engine

Tycho's temporal architecture provides a coherent framework for relating heterogeneous metric streams to fixed-duration analysis windows. It establishes a common time base, defines how collectors operate, and specifies how windows are formed and interpreted. The model is intentionally simple: collectors run independently, timestamps reflect poll time, and all temporal reasoning occurs during analysis.

4.4.1 Event-Time Model and Timestamp Semantics

Tycho adopts a single monotonic time base for all temporal coordination. Collectors timestamp each sample at the moment of observation; these timestamps reflect poll time, not the physical instant at which the underlying hardware event occurred. Event time is therefore a modelling construct used by the analysis engine when interpreting delay, freshness and update behaviour.

This separation keeps collectors lightweight and domain-agnostic. Each collector reports only what it directly observes; the analysis engine later interprets these timestamps in context, using calibration-derived delay characteristics to approximate underlying temporal structure.

4.4.2 Independent Collector Schedules

Tycho employs independent, domain-aware sampling schedules. During startup the timing engine configures one schedule per collector, after which each collector operates autonomously on its own periodic trigger. No global poll loop exists and collectors do not synchronise with one another. They push samples only when a new observation is available.

This decoupling avoids artificial temporal alignment and preserves each domain's intrinsic update behaviour. Collector timestamps are placed directly on the global monotonic time axis, allowing later reconstruction without imposing shared cadence or shared sampling semantics.

4.4.3 Window Construction and Analysis Triggering

Analysis proceeds in fixed-duration windows defined solely by periodic triggers from the timing engine. If the triggers occur at monotonic times $T_0, T_1, T_2, \ldots$, window W_i is the half-open interval $[T_i, T_{i+1})$. Window duration is nominally constant but may drift slightly, which is acceptable for attribution.

When a window closes, the analysis engine performs two conceptual phases:

(i) *idle characterisation*, using long-term buffered history across all relevant domains, and

(ii) *window reconstruction and attribution*, using all samples whose timestamps precede T_{i+1}.

Only energy for the current window is attributed and exported, but additional historical samples inform delay interpretation, idle estimation and interpolation.

Tycho treats domains asymmetrically: CPU and software metrics are always required; GPU and Redfish domains contribute when available. Samples too old to fall within the current window do not contribute directly but may still inform background characterisation. Windows remain valid when optional domains are absent.

A sample is considered stale relative to a window when its poll timestamp predates T_i by more than a domain-specific tolerance. Stale samples are ignored for direct reconstruction but do not invalidate the window.

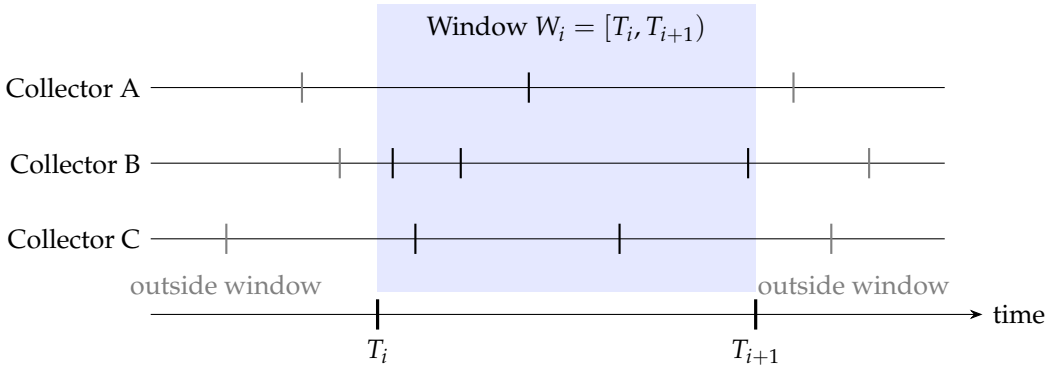


FIGURE 4.2: Analysis window W_i in relation to collectors

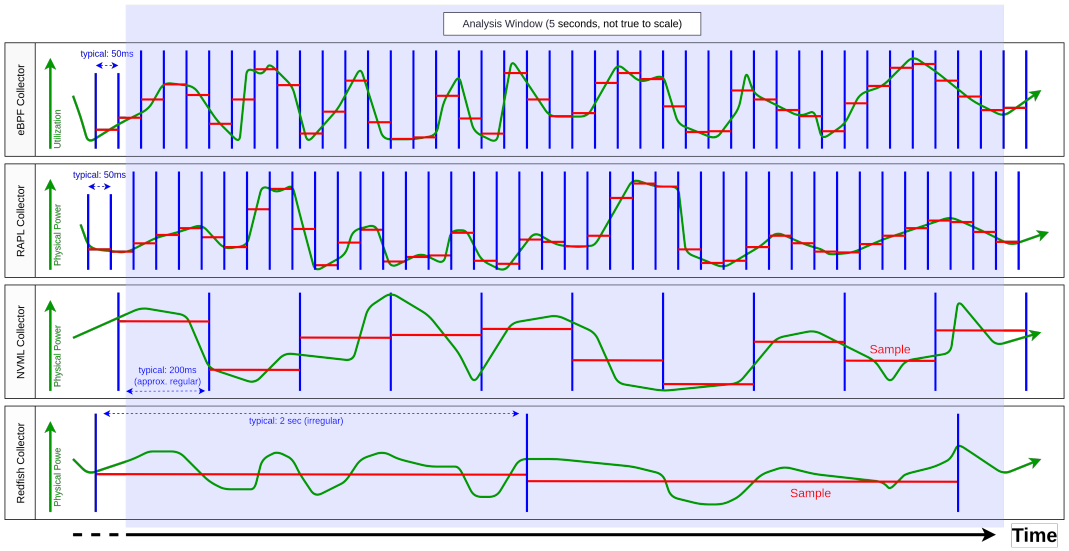
4.4.4 Comparison to Kepler Timing Model

Kepler employs a synchronous timing model in which all metric domains (except Redfish) are sampled within a single periodic poll cycle (default: 3 seconds). This fixed-length interval defines both the sampling cadence and the logical unit of attribution. Redfish updates occur at a much slower rate (default: 60 seconds), and the most recent Redfish value is reused across multiple attribution intervals. Export occurs on a separate cadence, which may not align with the attribution window.

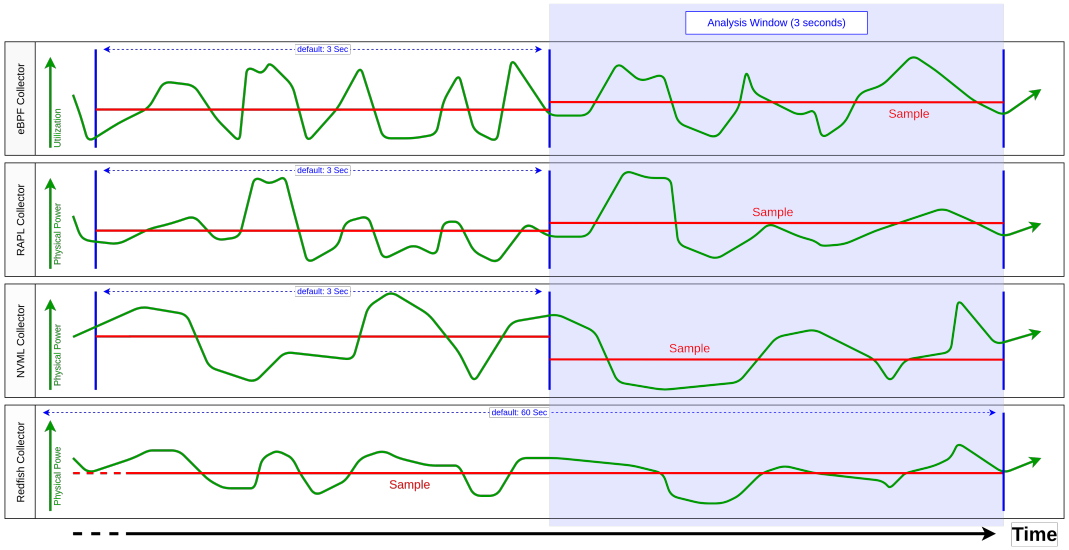
Tycho diverges fundamentally: collectors run independently, analysis windows are defined by attribution triggers rather than poll cycles, heterogeneous update patterns are supported natively, and export occurs immediately after each attribution step. This structure enables finer temporal resolution, avoids dependence on synchronous polling behaviour, and eliminates inconsistencies between data collection and publishing intervals.

Figures 4.3a and 4.3b illustrate the respective timing behaviour of Tycho and Kepler, highlighting their polling patterns, sampling semantics and analysis-window alignment. Figures 4.4a and 4.4b provide a higher-level view to show the Prometheus

export behaviour more clearly.

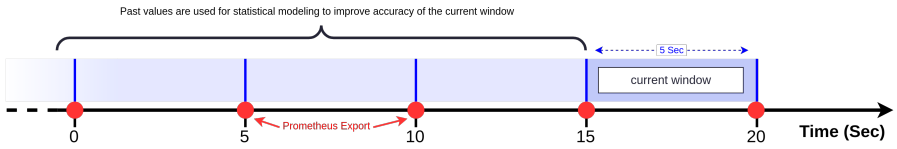


(A) Tycho Timing Model

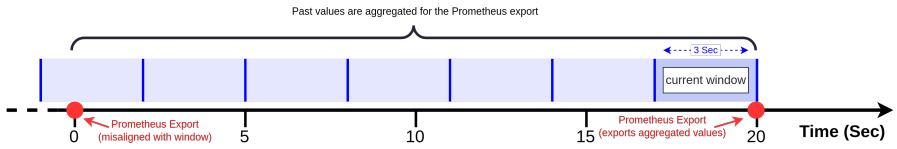


(B) Kepler Timing Model

FIGURE 4.3: Comparison: Tycho and Kepler Timing Model



(A) Tycho Prometheus Export Timing Model



(B) Kepler Prometheus Export Timing Model

FIGURE 4.4: Comparison: Tycho and Kepler export behaviour

4.5 Metric Sources as Temporal Actors

This section characterizes all metric sources as temporal actors that emit observations under distinct timing, latency, and reliability constraints. Rather than treating collectors as passive data providers, it formalizes their role as asynchronous producers whose outputs define the raw temporal structure available to the analysis pipeline. The following subsections describe the observable properties, guarantees, and limitations of each source, establishing the bounds within which subsequent metric construction and attribution operate.

4.5.1 eBPF and Software Counters

The eBPF and software counter domain represents Tycho's event-driven view of CPU activity. Unlike hardware domains that report values at fixed sampling times, this domain emits utilisation information at the moment execution state changes occur. These events form a temporally dense and workload-dependent signal that describes how processor time is distributed across user tasks, kernel execution, interrupt handling, and idle periods. All higher-level aggregation is performed in userspace.

The domain contributes three classes of metrics with distinct temporal semantics.

- *Event-driven metrics* capture transitions in processor ownership and record the exact time at which execution begins or ends for a given context.
- *Cumulative counters* accumulate activity or duration over time and expose their values when queried.
- *Quasi-instantaneous counters* sample hardware performance state at activity boundaries, with semantics tied to the execution periods they describe.

Because event timestamps directly encode execution boundaries, no domain-level delay calibration is required, and collector polling cadence does not affect temporal alignment. Each event carries container context at the point of observation, enabling correct attribution under workload migration across control groups.

Within Tycho's temporal model, this domain provides fine-grained ownership information at execution boundaries and yields a complete temporal partition of CPU activity within each analysis window. These signals support proportional attribution and reduce uncertainty in downstream energy modelling, making eBPF-derived utilisation the most temporally precise workload activity signal available to the system.

4.5.2 RAPL Domains

RAPL exposes cumulative energy counters for a set of logical CPU-related domains, including package, cores, uncore, and memory. Each domain provides a monotonically increasing counter that reflects total energy consumed since a hardware-defined reference point and advances independently of the sampling schedule.

Within Tycho, RAPL counters are observed at fixed tick boundaries. At each tick the current counter values are recorded, and interval energy follows from the difference between consecutive readings. RAPL therefore contributes energy over time rather than instantaneous power, with temporal resolution defined by the tick interval. Because hardware updates occur at a much higher rate than sampling, the counters behave as effectively continuous at the chosen time scale.

RAPL sampling is aligned with Tycho's timing engine such that each analysis interval contains exactly one cumulative reading per domain. Internal update behaviour is already integrated into the counters and does not affect interval attribution. Architecturally, RAPL acts as a stable and low-noise source of CPU-adjacent energy.

The domain structure of RAPL aligns naturally with Tycho's requirement for domain-level consistency. Per-socket counters for package, core, uncore, and memory domains form a coherent decomposition of CPU energy that is preserved across intervals and provides a reliable baseline against which software-side utilisation signals can be related during attribution.

4.5.3 Redfish/BMC Power Source

Redfish provides an out-of-band view of total node power through the server's Baseboard Management Controller. It reports instantaneous power values at coarse and implementation-defined intervals and constitutes Tycho's only system-wide power observation.

Within Tycho's architecture, Redfish is treated as a *latently published external observation* rather than as a synchronisable metric source. Sampling is performed at fixed tick boundaries using the global monotonic timebase, but temporal authority remains with the BMC. Repeated values are common, and new measurements may appear only after several ticks; Redfish therefore constrains temporal interpretation rather than defining it.

To make this uncertainty explicit, each Redfish observation is annotated with a *freshness* value that expresses the temporal distance between the BMC's reported update time, when available, and Tycho's collection time. Freshness is an architectural quality indicator rather than a correction mechanism and allows downstream analysis to reason about temporal reliability without assuming regular publication or low latency.

When no new BMC update appears for an extended period, Tycho emits an explicit continuation of the last known power value. Continuation samples preserve a complete and chronologically consistent power timeline while making the absence of new information explicit; they do not indicate new measurements.

Despite its limited temporal resolution, Redfish serves as Tycho's authoritative source for total node power. It anchors the system's global energy view and provides a stable reference against which CPU- and accelerator-level energy estimates can be interpreted, with its coarse publication behaviour accommodated through explicit timestamping, freshness annotation, and controlled continuation.

4.5.4 GPU Collector Architecture

Accelerators form a significant share of the power consumption of modern compute nodes. Tycho therefore integrates GPU telemetry into the same unified temporal framework that governs RAPL, Redfish, and eBPF sources. NVIDIA devices expose energy-relevant information only at discrete publish moments inside the driver, so GPU sampling cannot rely on periodic polling alone. Instead, Tycho aligns sampling with the device's internal update behaviour and publishes at most one `GpuTick` for each confirmed hardware update. All GPU ticks share the global monotonic timebase that underpins Tycho's event-time model (§ 4.4). This section specifies the architectural guarantees, temporal contracts, and admissible interpretations of GPU telemetry; numerical reconstruction techniques, solver behavior, and backend-specific realization details are treated as implementation concerns and are not part of the architectural contract.

Architectural Role. The GPU subsystem provides two forms of telemetry. Device-level metrics describe the instantaneous operating state of each accelerator, including power, utilisation, memory, thermals, and clock data. Process-level metrics describe backend-aggregated utilisation over a defined wall-clock window. Both streams are combined into a single `GpuTick` that represents the accelerator state at a specific moment in Tycho's global timeline. GPUs and MIG instances are treated as independent logical devices for the purpose of telemetry collection.

A central architectural design choice is the use of high-frequency *instantaneous* power fields exposed through NVIDIA's field interfaces, rather than relying exclusively on the conventional averaged power signal. Most existing GPU energy analyses depend on the one-second trailing average returned by `nvmlDeviceGetPowerUsage`, which obscures short-lived changes in power demand. By incorporating instantaneous power samples alongside averaged values, Tycho preserves substantially richer temporal structure at the telemetry source itself. This additional signal fidelity is a prerequisite for sub-second attribution and is later exploited by the analysis engine to improve temporal accuracy.

Backend Abstraction. The GPU collector interfaces with NVIDIA hardware through `NVML`, which provides access to device-level and process-level telemetry. The architecture introduces a backend abstraction layer to decouple the collector from a specific vendor interface. This abstraction permits alternative backends, such as `DCGM`, to be integrated in the future without altering the surrounding timing and buffering logic. In the current system, `NVML` is the sole implemented backend.

The architecture does not assume uniform telemetry availability across devices. Cumulative energy counters, instantaneous power fields, and process-level utilisation may or may not be exposed depending on GPU generation and configuration. These capability differences are treated as properties of individual devices and handled through per-device feature masks within the implementation.

Conceptual Sampling Model. GPU drivers update power and utilisation metrics at discrete, hardware-defined cadences that are not visible to callers. Polling at a fixed interval is fundamentally mismatched to this behaviour. If the polling frequency is lower than the internal publish cadence, updates are missed; if it is higher, the collector repeatedly observes identical values. Over time, this mismatch leads to aliasing, redundant samples, and temporal drift relative to other metric sources.

The sampling model distinguishes two conceptual modes. In base mode, the subsystem polls at a moderate frequency to track slow drift in the device's cadence. In *phase-aware sampling* mode, the subsystem temporarily increases its sampling frequency when Tycho's timebase approaches a predicted publish moment. This concentrates sampling effort where a fresh update is expected and reduces latency between the hardware update and Tycho's observation of it. As a result, a new sample can be detected earlier (and hence, with a more accurate timestamp), while avoiding additional overhead introduced by constant hyperpolling. The architecture guarantees that sampling remains event-driven rather than periodic, as formalised by the phase-aware timing model.

Formal Timing Model. The GPU collector relies on a phase-aware timing model to align sampling with the implicit publish cadence of the device driver. Because this cadence is not exposed by the hardware or backend interface, it must be inferred from observed updates and expressed relative to Tycho's global monotonic timebase.

Let $t_{\text{obs},k}$ denote the monotonic timestamp of the k-th confirmed GPU publish event. Successive observations define inter-update intervals $\Delta t_k = t_{\text{obs},k} - t_{\text{obs},k-1}$, which serve as samples of the device's publish period. The model maintains a smoothed period estimate $\hat{T}$ and a phase offset $\hat{\phi}$ that jointly predict the timing of future publishes.

At any time t, the predicted next publish moment t_{next} is obtained by advancing the most recent observation by an integer multiple of $\hat{T}$, adjusted by $\hat{\phi}$, such that $t_{\text{next}} \geq t$. Sampling effort is concentrated in a narrow window around t_{next}, while lower-frequency polling maintains coarse alignment and tracks long-term drift.

This model establishes the following architectural guarantees:

- GPU sampling is aligned to inferred publish events rather than to a fixed polling interval.
- Each hardware publish produces at most one logical GPU event.
- No GPU event is emitted without a detectable device update.

The model is intentionally agnostic to backend-specific mechanisms used to detect freshness or to refine period and phase estimates. These concerns are delegated to

the implementation, which must realise the model under partial observability, jitter, and backend variability while preserving the guarantees above.

Figure 4.5 illustrates this behaviour at the architectural level, showing the relationship between the GPU's implicit publish events, Tycho's adaptive polling activity, and the resulting sequence of emitted `GpuTick`s.

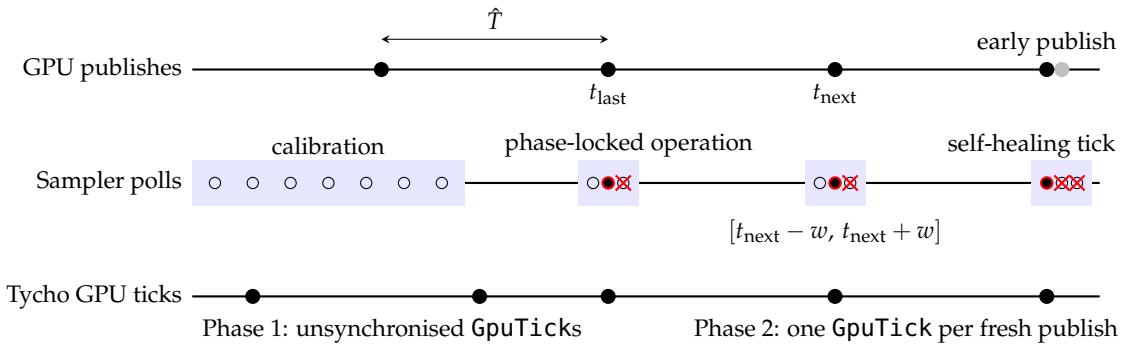


FIGURE 4.5: Phase-aware GPU polling timeline

Tick Semantics. A `GpuTick` is emitted only when Tycho detects a genuinely new hardware update. Each tick contains a snapshot of device-level metrics and, when available, process-level utilisation aligned to the same monotonic timestamp. This design ensures that GPU measurements participate in Tycho's cross-domain correlation without interpolation, resampling, or ad hoc realignment. The one-to-one correspondence between hardware updates and GPU ticks is a core architectural guarantee and the primary distinction between Tycho's approach and traditional periodic sampling.

Process Telemetry Integration. Process-level metrics describe aggregated utilisation over a wall-clock window that is defined by the backend rather than Tycho's timing engine. The architecture treats these windows as retrospective measurements that must be aligned with the device timeline. Each process record is anchored to the timestamp of the device snapshot that triggered its acquisition. This preserves temporal coherence in spite of the retrospective semantics of process telemetry and supports multi-tenant attribution across GPU workloads.

Integration with the Global Timing Model. All GPU ticks are timestamped using Tycho's global monotonic timebase and inserted into the multi-domain ring buffer. This ensures strict temporal ordering relative to RAPL, Redfish, and eBPF data. The architecture maintains the principle of domain autonomy: each subsystem generates updates according to its own temporal behaviour, while the analysis engine later fuses these streams into a consistent attribution result.

Architectural Limitations. Although the architecture abstracts over backend differences, several structural constraints remain. Telemetry capabilities vary significantly across NVIDIA devices and driver configurations. Some accelerators expose high-quality instantaneous power fields and cumulative energy counters, while others provide only averaged power and coarse utilisation.

The implicit publish cadence may drift under DVFS or thermal transitions, which limits the predictability of update edges. Tycho mitigates these effects through robust sampling logic in the implementation, but the fidelity of the resulting GPU timeline remains bounded by the behaviour of the underlying hardware.

Overall, the GPU subsystem elevates accelerator telemetry to a first-class component of Tycho's energy model. By aligning sampling with the device's publish behaviour and unifying device and process metrics under a single timestamping model, the architecture enables precise, temporally consistent attribution in heterogeneous accelerator environments.

4.6 Metadata Collection Subsystem

Tycho treats workload identity as a first-class architectural concern that is strictly separated from numerical telemetry. While energy and utilisation collectors emit temporally ordered measurement streams, metadata captures the structural relationships required to interpret those streams during analysis. This includes the association of processes, containers, and pods as they evolve over the lifetime of a node.

Metadata is maintained as cached identity state rather than as a time series. It is neither aggregated nor iterated over analysis windows and does not participate directly in temporal correlation. Instead, metadata provides a bounded, continuously refreshed snapshot of recent workload structure that must remain sufficiently fresh and temporally consistent to support later attribution. Consequently, metadata collection prioritises controlled refresh and bounded lifetime over high-frequency or event-level precision.

Subsystem overview. The metadata subsystem forms a dedicated architectural layer that operates independently of metric collection and analysis. It consists of a small set of autonomous collectors coordinated through a single metadata controller, which constitutes the sole authority over metadata mutation and lifecycle management. Collectors observe the system independently, but all state updates are mediated by the controller, enforcing a clear separation between identity acquisition and subsequent analytical processing.

Dual-collector model. Workload identity is inherently multi-sourced. Tycho therefore integrates two complementary metadata collectors, each providing a partial and independently valid view of system state:

- **proc-collector:** observes process identity, execution context, and cgroup membership directly from the Linux kernel via the filesystem interface, providing authoritative runtime state independent of orchestration abstractions.
- **kubelet-collector:** acquires namespace-, pod- and container-level identity from the Kubernetes node agent, exposing scheduling and lifecycle information unavailable at the operating-system level.

The metadata subsystem does not attempt to fuse or interpret these views at collection time. Instead, it records the most recent identity state observed by each source and defers reconciliation to analysis-time logic.

Controller-based coordination and scheduling. Metadata collection in Tycho is explicitly *analysis-driven*. The start of an analysis cycle constitutes the primary trigger for metadata refresh and is treated as the highest-priority collection opportunity. When an analysis window begins, the analysis engine requests a best-effort refresh of all registered metadata collectors in order to obtain the most recent possible identity state.

Autonomous metadata collectors exist as a secondary mechanism whose sole purpose is to bound metadata age when analysis intervals are long. These collectors execute under controller supervision and are explicitly subjugated to analysis-driven collection. If an analysis-triggered refresh is imminent, periodic collectors are suppressed and defer execution to the analysis engine, preventing redundant collection in short succession.

All collectors register with a central metadata controller, which arbitrates between analysis-triggered and background execution. The controller tracks the timestamp of the most recent successful observation for each collector and enforces source-specific freshness constraints. Collection is permitted only when required to satisfy source-specific freshness constraints. As a result, metadata may be refreshed multiple times within a single analysis window under long-running analysis, while redundant collection near analysis boundaries is suppressed to reduce overhead.

This prioritisation establishes a clear architectural guarantee. Metadata is maximally fresh at analysis start, redundant collection is avoided, and collection overhead remains bounded independently of both analysis frequency and global scheduling cadence.

Metadata state and lifetime model. Collected metadata is stored in a dedicated in-memory state that represents a bounded snapshot of recent workload identity. Unlike the ring-buffer–based design used for metric data, the metadata store retains only the most recent valid representation of each observed entity. Entries correspond to identity-bearing objects such as processes, containers, and pods and are keyed by stable identifiers. New observations update entries in place; historical versions and event sequences are not preserved beyond the lifetime of the buffer window.

Each metadata entry carries a monotonic timestamp anchored to Tycho's global timebase, allowing identity state to be interpreted consistently alongside energy and utilisation measurements. Metadata is considered valid from its most recent observation until it is removed by lifecycle management. Garbage collection is horizon-based and enforced exclusively by the controller, which removes entries once they fall outside the retained temporal window. Collectors never delete metadata directly, ensuring deterministic expiry and consistent memory bounds.

By coupling metadata lifetime to a bounded horizon rather than to explicit lifecycle events, the subsystem remains robust to incomplete or delayed observations. Terminated processes, containers, and pods persist only long enough to support analysis windows and are removed automatically once they passt the removal horizon.

4.7 Calibration

Calibration is an auxiliary architectural subsystem that bounds temporal uncertainty introduced by hardware-controlled metric publication. It exists to constrain polling behaviour and temporal alignment for metric sources whose update cadence or observable reaction latency is externally governed and not analytically predictable. Calibration is applied selectively and only where such uncertainty significantly affects the correctness of subsequent analysis.

Calibration produces static, conservative parameters that are consumed by the timing and analysis subsystems. It does not participate in runtime attribution, does not adapt dynamically, and does not operate on live metric streams. By resolving temporal uncertainty ahead of time, calibration allows the runtime system to remain deterministic, bounded, and non-intrusive.

Tycho distinguishes two independent calibration concerns: *polling-frequency calibration*, which bounds how often a metric source must be queried to avoid undersampling hardware updates, and *delay calibration*, which bounds the latency between a workload transition and the first observable reaction in a metric stream. These concerns are orthogonal and are applied only where their respective assumptions hold.

Polling-frequency calibration. Polling-frequency calibration applies to metric sources whose publish cadence is hardware-controlled and approximately regular. Its purpose is to derive a conservative polling interval that observes all published updates under nominal conditions without imposing unnecessary collection overhead.

Polling-frequency calibration is performed during Tycho startup. It relies exclusively on passive observation of device behaviour and does not require workload manipulation. This calibration is applied to GPU and Redfish power metrics, whose firmware- or BMC-controlled publication intervals are stable in expectation but not analytically documented. The resulting polling bounds are treated as configuration constraints by the timing subsystem and remain fixed during normal operation. For node-level execution, Tycho adopts the most conservative bound across all contributing devices to ensure uniform temporal coverage.

Polling-frequency calibration is not applied to RAPL or eBPF. RAPL energy counters update quasi-continuously at a granularity far below Tycho's sampling resolution, rendering undersampling architecturally irrelevant. eBPF metrics are event-driven and decoupled from device-side publish cadence, making polling-frequency discovery unnecessary.

Delay calibration. Delay calibration bounds the latency between a workload transition and the first observable change in a metric stream. This calibration applies only where such latency is stable, workload-independent, and sufficiently repeatable to be treated as a bounded constant.

Delay calibration is performed exclusively for GPU power metrics. GPU devices internally aggregate and buffer power readings prior to publication, introducing a measurable and consistent delay relative to workload onset. Accurate estimation of this delay requires the generation of controlled, high-intensity workload transitions to elicit clear device responses. As Tycho is architecturally constrained to non-intrusive observation, such stimulus-driven measurement is performed offline and

excluded from runtime operation. The resulting delay bounds are supplied as static configuration parameters and inform the analysis subsystem's temporal alignment logic without participating in runtime attribution.

No delay calibration is performed for RAPL or eBPF. At Tycho's temporal resolution, residual access latency in RAPL energy counters is negligible, and eBPF metrics reflect execution state transitions without device-side buffering. Both domains are therefore treated as temporally immediate at the architectural level.

Delay calibration is not applied to Redfish. Redfish power readings exhibit irregular publish intervals, variable network latency, and opaque BMC-internal behaviour, precluding stable delay estimation. Redfish metrics are consequently treated as coarse, low-resolution signals suitable for slow global trends, with temporal consistency enforced through separate freshness and scheduling mechanisms. Architecturally, calibration constrains admissible interpretation without introducing additional runtime state or adaptive behavior.

4.8 Analysis and Attribution Architecture

4.8.1 Pipeline Orchestration and Stage Execution

4.8.1.1 Problem Statement

Tycho's analysis layer must transform heterogeneous, asynchronous observations into window-scoped attribution results under three constraints. Inputs originate from independent sources with bounded but non-uniform delays and may be temporarily unavailable. Attribution further requires interdependent derived quantities, imposing strict ordering constraints. Finally, analysis must remain online, deterministic, and non-retrospective, excluding retrospective reinterpretation and any semantic coupling to exporter behavior. The orchestration problem is therefore to execute a deterministic, dependency-respecting transformation pipeline per attribution window while tolerating partial observability and enforcing the invariants defined in § 4.4.3.

4.8.1.2 Conceptual Model

Analysis proceeds as a sequence of discrete *cycles*. Each cycle selects a single attribution window, instantiates a self-contained execution context, executes a fixed set of transformations in a predefined order, and yields a logically atomic set of window-scoped derived quantities. The analysis engine acts solely as an orchestration authority. It determines the temporal scope of each cycle using the global monotonic timebase, enforces execution order, and commits results as a coherent unit. Collection, buffering, and metadata acquisition are upstream concerns and are assumed to have materialized raw observations into bounded-retention histories.

The pipeline is expressed as a set of *metrics*, each representing a typed transformation. Metrics consume raw observations or previously derived metrics and emit window-scoped results according to the semantic models defined in this chapter. Metrics may depend on earlier outputs within the same cycle but never on downstream publication state or future cycles.

4.8.1.3 Attribution Window Semantics

Let t_k denote the monotonic timestamp associated with the start of analysis cycle k. The attribution window for cycle k is defined as the half-open interval

$$W_k = (t_{k-1}, t_k]. \tag{4.1}$$

Windows are defined on a global monotonic timebase, form a total order, and are non-overlapping.

Window selection incorporates a fixed intentional lag relative to real-time execution. The window end t_k is chosen such that it lags the most recent observations by at least the maximum admissible metric delay plus a safety margin, derived from configuration and optional calibration (§ 4.7). This bound is an architectural precondition for window validity and guarantees that all metrics participating in a cycle can interpret their contributions over W_k under their declared delay semantics. As a result, attribution correctness is decoupled from collector jitter, speculative window closure is avoided, and causality is preserved.

In steady state, attribution windows have a fixed duration. During startup, when insufficient history exists, the window start may be clamped to the beginning of the monotonic timeline. This is the only permitted deviation from the steady-state definition and preserves determinism.

4.8.1.4 Stage Model and Dependency Discipline

Analysis is structured as an ordered sequence of conceptual stages reflecting semantic dependencies between derived quantities. A stage delineates computations whose outputs serve as prerequisites for subsequent modeling or attribution steps. Within a cycle, stages execute strictly in order. Metrics may depend on raw observations and on outputs from earlier stages in the same cycle, but must not depend on later stages, future cycles, or sink side effects.

This discipline renders the pipeline compositional. Later attribution logic operates on materialized, window-scoped quantities rather than ad hoc joins over raw histories. Stages are semantic boundaries, not runtime entities, and exist to make dependency structure explicit when partial observability yields incomplete but internally consistent results.

4.8.1.5 Best-Effort Semantics Under Partial Observability

The analysis architecture is accuracy-first but not completeness-first. For a given window, Tycho produces the most complete set of derived quantities that can be computed without violating architectural invariants. If required inputs are unavailable or inadmissible under a metric's semantics, that metric is undefined for the window and does not materialize a result. Downstream metrics may still execute if their dependencies are satisfied, yielding partially populated but self-consistent outputs. As observability decreases, results degrade monotonically through omission or explicitly defined fallback semantics, and previously valid interpretations are never revised.

4.8.1.6 Output Commit and Sink Boundary

All results produced during a cycle belong to the same attribution window W_k and are committed as a logical batch. Sinks are strictly downstream observers and lie outside the correctness boundary of attribution. Exporter behavior may delay or drop publication but does not affect the meaning of computed window-scoped quantities. This separation prevents exporter mechanics from becoming an implicit part of attribution semantics and preserves reproducibility.

4.8.1.7 Architectural Consequences

The orchestration model establishes a stable execution contract. Stage-local computations may assume a well-defined attribution window, deterministic ordering, and read-only access to upstream histories. In return, analysis is constrained to be window-scoped and non-retrospective. Each cycle yields a single, maximal, internally consistent interpretation of the evidence available for its window, and later cycles do not revise earlier results. This contract supports the staged construction of increasingly sophisticated attribution models without altering orchestration semantics.

4.8.2 Stage 1: Component Metric Construction

This stage defines how raw observations emitted by the collectors are transformed into coherent per-component total energy and power metrics. Its purpose is to establish temporally aligned, conservation-preserving component signals from heterogeneous inputs, independent of any attribution or decomposition semantics. The resulting metrics form the authoritative total-energy basis for all subsequent stages and are exported as externally consumable component-level measurements.

4.8.2.1 Component-Level eBPF Utilization Metrics

Problem Statement. eBPF exposes raw kernel execution signals and hardware event counts as per-tick deltas. These signals must be transformed into window-aligned utilization metrics with well-defined semantics and into cumulative counters suitable for direct export and downstream aggregation.

Conceptual Model. Two classes of node-level metrics are constructed from eBPF observations. CPU execution time is expressed as normalized time-share ratios over a fixed analysis window. All other kernel and hardware signals are expressed as cumulative counters obtained by aggregating per-process deltas across the node.

Formalization. Let a node expose C logical CPUs and let an analysis window of duration Δt define a total schedulable capacity

$$T_{\text{cap}} = C \cdot \Delta t. \tag{4.2}$$

Let T_{idle}, T_{irq}, and T_{softirq} denote the aggregated CPU time spent in idle, hardware interrupt, and software interrupt execution over the window. Normalized utilization ratios are defined as

$$r_x = \frac{T_x}{T_{\text{cap}}}, \quad x \in \{\text{idle}, \text{irq}, \text{softirq}\}. \tag{4.3}$$

Active execution is defined as the residual

$$r_{\text{active}} = 1 - (r_{\text{idle}} + r_{\text{irq}} + r_{\text{softirq}}). \tag{4.4}$$

By construction, all ratios are dimensionless, bounded in $[0, 1]$, and satisfy

$$r_{\text{idle}} + r_{\text{irq}} + r_{\text{softirq}} + r_{\text{active}} = 1. \tag{4.5}$$

For each kernel or hardware event type j, let ΔK_j denote the per-tick delta aggregated across all processes. The exported counter is defined as the cumulative sum

$$N_j(t) = \sum_{k \le t} \Delta K_j, \tag{4.6}$$

which is monotonically non-decreasing over the lifetime of the process.

Design Decisions. CPU utilization is normalized to node capacity to make ratios invariant to window length and core count. Active CPU time is defined as a residual to enforce exact conservation of CPU capacity. All event-based metrics are exposed as cumulative counters to preserve monotonicity and allow rate derivation without reinterpreting window semantics.

Architectural Consequences. The resulting metrics provide window-stable CPU utilization ratios and strictly monotonic kernel activity counters. These quantities can be consumed directly as observability metrics and reused unchanged by downstream attribution stages.

4.8.2.2 RAPL Component Metrics

Problem Statement. RAPL exposes cumulative energy counters per hardware domain, but these counters are node-local, socket-scoped, and offset by an arbitrary hardware-defined origin. For Tycho, these signals must be transformed into a single, coherent component-level energy metric that is comparable across time and suitable as an authoritative input for later attribution stages. The core problem is therefore to define a metric that preserves the physical meaning of RAPL energy while eliminating hardware-specific offsets and remaining stable under windowed analysis.

Conceptual Model. The RAPL component is modeled as a set of independent energy domains, treated as conceptually separate devices but belonging to the same component. For each domain, Tycho constructs a cumulative energy signal that represents the total physical energy consumed since Tycho start. This signal is defined independently of any windowing semantics and serves as the primary representation of RAPL energy within the system. A secondary, auxiliary power signal is derived from the same observations for user-facing inspection, but it is not authoritative.

Formalization. Let $E_d^{\text{raw}}(t)$ denote the raw cumulative RAPL energy reading for domain d at time t, as provided by the underlying measurement subsystem. These raw counters are assumed to be strictly monotonic and already corrected for hardware wraparound.

For each domain d, Tycho defines an exported cumulative energy counter

$$E_d(t) = E_d^{\text{raw}}(t) - E_d^{\text{raw}}(t_0), \tag{4.7}$$

where t_0 is the time of the first observed sample for that domain. This construction yields a zero-based, monotonic energy counter with $E_d(t_0) = 0$.

An auxiliary average power signal is defined over an analysis window $[t_i, t_{i+1}]$ as

$$P_d^{(i)} = \frac{E_d(t_{i+1}) - E_d(t_i)}{t_{i+1} - t_i}. \tag{4.8}$$

This quantity is derived solely from the cumulative energy counter and has no independent physical authority.

Design Decisions. The primary design choice is to treat cumulative energy as the first-class metric and to derive all other quantities from it. Using zero-based cumulative counters removes dependence on hardware-specific initial offsets and simplifies downstream reasoning. Power is intentionally modeled as a derived, window-local quantity rather than as a primary signal, reflecting its lower robustness and its lack of necessity for attribution. Domains are defined once and treated uniformly, avoiding domain-specific special cases in the metric definition.

Architectural Consequences. The exported RAPL energy counters form the authoritative energy input for all later attribution and decomposition stages. Their monotonicity and zero-based semantics allow unambiguous differencing, aggregation, and conservation reasoning. By contrast, the power metrics are explicitly auxiliary, non-authoritative, and excluded from downstream attribution logic. This separation ensures that attribution correctness depends only on cumulative energy, while still permitting power-oriented inspection for diagnostic or user-facing purposes.

4.8.2.3 GPU-Corrected Energy Metric

Problem Statement. GPU telemetry provides multiple power- and energy-related observations of the same physical process under incompatible temporal semantics. Instantaneous power samples, one-second averaged power values, and optional cumulative energy counters are reported concurrently but cannot be combined by direct integration or signal selection without introducing temporal bias or discarding information. Using any single signal in isolation either preserves absolute correctness at coarse granularity or improves temporal resolution at the cost of systematic error. The problem addressed here is therefore not measurement scarcity, but the absence of a principled method to reconcile redundant GPU observations into a single, temporally refined, internally consistent power and energy signal aligned to corrected time.

Conceptual Model. GPU power is modeled as a latent, non-negative, continuous-time signal reconstructed on a uniform corrected-time grid and maintained over a retained history horizon that exceeds the current analysis window. This deliberate use of historical context is a central accuracy mechanism, allowing delayed, coarse, and partially redundant observations to be reconciled in a globally consistent manner that window-local estimation cannot achieve. All raw GPU observations are

interpreted as constraints on this latent signal, jointly shaping a single authoritative power timeline per device. Instantaneous and one-second averaged power samples impose soft consistency constraints, while cumulative energy counters, when available, act as dominant anchors that strongly influence the reconstruction without enforcing hard equality. Physical plausibility is enforced by penalizing implausible curvature while preserving total energy and long-term magnitude. Windowed GPU energy and power metrics are obtained by integrating sub-intervals of this maintained reconstructed signal, ensuring temporal coherence across windows and improved accuracy relative to any raw observation stream.

Formalization. For each GPU device, power is represented by a reconstructed sequence $p = \{p_k\}_{k=0}^{N-1}$ on a uniform corrected-time grid with spacing Δt. Reconstruction is posed as a constrained optimization problem that minimizes weighted discrepancies to observed telemetry while enforcing physical plausibility. Let $\mathcal{R}$ denote the set of observation constraints derived from instantaneous power samples, one-second averaged power samples, and optional cumulative energy readings. Each constraint $r \in \mathcal{R}$ is expressed by a linear operator a_r acting on p with target value y_r and weight w_r.

The architectural objective is

$$\min_{p \geq 0} \sum_{r \in \mathcal{R}} w_r^2 \left(a_r^\top p - y_r \right)^2 + \lambda_D \sum_{k=1}^{N-2} \left(p_{k+1} - 2p_k + p_{k-1} \right)^2, \tag{4.9}$$

where the second-difference term penalizes curvature without imposing a prior on absolute magnitude. Non-negativity $p_k \geq 0$ enforces physical feasibility. Cumulative energy observations contribute constraints whose weights dominate when present, prioritizing energy consistency over time without enforcing exact equality. The solution p defines the authoritative GPU power signal; windowed GPU energy is obtained by integrating p over the corresponding corrected-time interval.

Design Decisions. Constrained reconstruction is chosen over signal selection to avoid privileging any single telemetry source and to preserve all available information. Cumulative energy counters are treated as dominant but soft constraints to exploit their reliability while tolerating gaps, resets, and delayed reporting. Instantaneous and averaged power samples are retained as complementary observations that improve temporal resolution and stabilize reconstruction when energy counters are absent. Smoothness is imposed via second differences to suppress implausible oscillations without biasing total energy or sustained power level. No magnitude prior is introduced, as any bias toward lower power would conflict with the accuracy-first objective. Numerical conditioning and solver-specific stabilization are intentionally excluded from the architectural model.

Architectural Consequences. The architecture yields a single corrected GPU power signal per device that is temporally finer than any raw input and internally consistent across power and energy representations. All downstream GPU metrics are derived exclusively from this reconstructed signal, eliminating ambiguity from heterogeneous telemetry. Historical retention becomes a first-class correctness mechanism, enabling stable, coherent windowed estimates under delayed and coarse observation regimes. The design explicitly acknowledges the modeled nature of the result while guaranteeing physical plausibility and maximal energy consistency given

available observations, establishing a stable foundation for subsequent attribution and aggregation stages.

4.8.2.4 Redfish-Corrected System Energy Metric

Problem Statement. Raw Redfish power telemetry is sparse, irregular, and subject to delay that can vary over time. Direct window integration of held power values yields a valid low-rate estimate, but it cannot provide temporally granular system power that is consistent across windows when sample timing drifts. A second construction is therefore required that treats Redfish as the authoritative system-level anchor while reconstructing a higher-rate system power trajectory from contemporaneous component-proxy signals.

Conceptual Model. The metric family exports a canonical system power series and its cumulative energy counter, parameterized by a 'source' label. During warmup or when insufficient Redfish anchoring is available, `source="redfish_raw"` is formed by integrating the held Redfish power trajectory. Once sufficient Redfish observations exist over a reconstruction horizon, `source="redfish_corrected"` replaces the raw series and represents a reconstructed system power trajectory on a fine time grid. Reconstruction is posed as fitting a non-negative linear combination of proxy signals that are expected to co-vary with system power, while strongly anchoring the fit to the observed Redfish samples.

Formalization. Let $p_{\mathrm{RF}}(t)$ denote Redfish-reported system power (in mW), observed at irregular raw times t_j^{raw}. Redfish observations are subject to an unknown, time-varying reporting delay. The construction therefore introduces an explicit delay parameter $\delta_{\mathrm{RF}} \geq 0$, mapping raw timestamps into a corrected domain

$$t_j = t_j^{\mathrm{raw}} - \delta_{\mathrm{RF}}. \tag{4.10}$$

For the raw series, $p_{\mathrm{raw}}(t)$ is defined as a zero-order-hold trajectory of $p_{\mathrm{RF}}(t)$ in corrected time. For an analysis window $W = [t_s, t_e]$ of duration Δ_W seconds, the window energy and average power are

$$E_{\mathrm{raw}}(W) = \int_{t_s}^{t_e} p_{\mathrm{raw}}(t)\, dt, \qquad P_{\mathrm{raw}}(W) = \frac{E_{\mathrm{raw}}(W)}{\Delta_W}. \tag{4.11}$$

For the corrected construction, system power is reconstructed on a uniform grid of bins k with width Δ milliseconds over a finite horizon. Let $p_{\mathrm{hat}}[k]$ denote reconstructed system power in bin k. For each bin, proxy features are derived from contemporaneous component metrics: average package, DRAM, and GPU powers $p_{\mathrm{pkg}}[k]$, $p_{\mathrm{dram}}[k]$, $p_{\mathrm{gpu}}[k]$, and the CPU instruction rate $r_{\mathrm{instr}}[k]$.

Redfish observations are projected into the corrected domain at times t_j and interpreted as constraints on the reconstruction (optionally corresponding to a trailing-average kernel over a fixed interval). Let y_j denote the Redfish power value associated with observation j. Reconstruction fits parameters $\theta = (\alpha, \beta, \gamma, \delta, b)$ such that

$$p_{\mathrm{hat}}[k] = \max\big(0,\, \alpha\, p_{\mathrm{pkg}}[k] + \beta\, p_{\mathrm{dram}}[k] + \gamma\, p_{\mathrm{gpu}}[k] + \delta\, r_{\mathrm{instr}}[k] + b\big)\,, \tag{4.12}$$

where $\alpha, \beta, \gamma, b \geq 0$ enforce physical non-negativity.

Crucially, δ_{RF} is not fixed. For each analysis cycle, a finite candidate set of delays is evaluated, and the delay is selected that minimises the unexplained system power relative to the proxy sum over recent bins, subject to stability constraints. This adaptive delay selection aligns Redfish observations to the proxy domain in a best-effort sense and is recomputed as system behaviour changes.

Window energy and power for the corrected series are obtained by integrating p_{hat} over the overlap of W with the reconstruction grid:

$$E_{\mathrm{corr}}(W) = \int_{t_s}^{t_e} p_{\mathrm{hat}}(t)\, dt, \qquad P_{\mathrm{corr}}(W) = \frac{E_{\mathrm{corr}}(W)}{\Delta_W}. \tag{4.13}$$

Design Decisions. Redfish is treated as the system-level anchor rather than as a weak label, so the fit objective is defined in observation space and evaluated only where Redfish provides constraints. Delay is treated as an explicit degree of freedom in the construction because raw Redfish timestamps are not sufficient to guarantee stable alignment. Proxy features are restricted to quantities that are already available at fine granularity, enabling reconstruction without introducing additional sensors. Non-negativity constraints on physically interpretable coefficients prevent the model from compensating Redfish irregularities by introducing negative component contributions or a negative baseline.

Architectural Consequences. The construction yields a single canonical system series that remains available during warmup via `source="redfish_raw"` and transitions to a higher-rate anchored estimate via `source="redfish_corrected"` without changing metric IDs. Downstream stages can treat the corrected series as the system-level power baseline for further decomposition, while retaining the provenance of the construction through the 'source' label. The reconstruction is intentionally conservative: when anchoring observations are insufficient or alignment is unreliable, corrected output is withheld and raw integration remains authoritative.

4.8.3 Stage 2: System-Level Energy Model and Residual

Problem Statement. After Stage 1, Tycho provides multiple component-level energy signals that are individually conservative but incomplete. No combination of RAPL and GPU measurements can fully account for total node energy consumption, and a system-level reference is required to constrain attribution. However, the only available system-wide signal, Redfish-reported power, exhibits limited and variable temporal fidelity. This creates a fundamental tension between physical conservation and temporal alignment that cannot be resolved by algebraic refinement alone.

Conceptual Model. Stage 2 introduces a system-level energy balance scoped to a single node and a single analysis window. Total system energy is taken from the Redfish-integrated signal, while accounted component energy is defined as the sum of all modeled contributors. Any remaining energy is captured explicitly as a residual term, which represents unmodeled components and temporal mismatch rather than error. Residual energy is therefore treated as a first-class quantity within the attribution pipeline.

Formalization. Let E_{sys} denote total system energy for a window, obtained from the system-level source. Let E_{parts} denote the sum of all accounted component energies for the same window. The residual energy E_{res} is defined as:

$$E_{\text{res}} = E_{\text{sys}} - E_{\text{parts}}. \tag{4.14}$$

This definition is local to a node and window and does not imply workload attribution. The architectural invariant enforced by Stage 2 is strict energy conservation at the window level.

Design Decisions. Residual energy is modeled explicitly rather than absorbed into noise or redistributed across components. This avoids introducing hidden assumptions about unmodeled hardware behavior and preserves auditability. Temporal misalignment between system and component signals is tolerated at the power level but never allowed to violate energy conservation. No attempt is made at this stage to reinterpret or smooth the residual term.

Architectural Consequences. Stage 2 establishes a closed system-level energy budget that constrains all downstream attribution. Later stages may further decompose or attribute residual energy, but they cannot eliminate it without introducing additional assumptions. The explicit residual also enables the system to signal when residual-based interpretation is unreliable, without compromising conservation.

4.8.4 Stage 3: Idle and Dynamic Energy Semantics

This stage defines how total component energy is partitioned into idle and dynamic contributions prior to attribution. Because CPU packages, residual system energy, and GPUs differ in observability, baseline stability, and utilization coupling, the decomposition semantics are specified per component rather than uniformly. The following subsubsections formalize these component-specific rules while preserving energy conservation and temporal consistency, and while establishing dynamic signals suitable for downstream attribution. Each decomposition defined below follows the same architectural pattern: a conservative baseline definition, a non-negative dynamic remainder obtained by conservation, and explicit admissibility conditions that bound when the resulting quantities may be interpreted.

4.8.4.1 RAPL Idle and Dynamic Decomposition

Problem Statement. RAPL exposes per-domain total energy that merges baseline platform consumption with workload-induced variation. For attribution and downstream fairness policies, Tycho requires a conservative split into an approximately stable idle component and a residual dynamic component while preserving per-window energy conservation.

Conceptual Model. For each RAPL domain $d \in \{\texttt{pkg}, \texttt{core}, \texttt{uncore}, \texttt{dram}\}$, Tycho treats the exported total power $P_d^{\text{tot}}(t)$ as the sum of an idle baseline $P_d^{\text{idle}}(t)$ and a dynamic remainder $P_d^{\text{dyn}}(t)$. Idle is estimated from low-activity operating points using an external utilization proxy $u_d(t)$ and a model that targets the theoretical $u_d = 0$ intercept, avoiding reliance on observing a true zero-load system. This preference follows the empirical finding that linear models yield robust idle estimates while higher-order fits tend to be unreliable in practice [52].

Formalization. Let W_k denote an analysis window of duration Δt_k and let $\Delta E_{d,k}^{\text{tot}}$ be the total domain energy increment over W_k. Tycho defines the decomposition by a baseline power estimate $\beta_{d,k}$ and constructs window energies as:

$$\Delta E_{d,k}^{\text{idle}} = \min\left(\Delta E_{d,k}^{\text{tot}},\ \max(0, \beta_{d,k}) \cdot \Delta t_k\right), \tag{4.15}$$

$$\Delta E_{d,k}^{\text{dyn}} = \Delta E_{d,k}^{\text{tot}} - \Delta E_{d,k}^{\text{idle}}. \tag{4.16}$$

The baseline $\beta_{d,k}$ is obtained by fitting a linear model $P_d^{\text{tot}} \approx \alpha_d u_d + \beta_d$ using only samples drawn from stable, low-utilization regimes and evaluating the intercept β_d as the idle estimate.

Design Decisions. Tycho adopts utilization-conditioned estimation rather than pure lower-bound tracking to remain meaningful on continuously active Kubernetes nodes where true idle is rarely observed. Model fitting is restricted to low-utilization operating points to prioritise identifiability of the intercept and avoid distortion by high-load regimes. The decomposition is defined per RAPL domain, permitting distinct baselines for domains whose activity sensitivity differs. Conservative clamping enforces non-negativity and prevents idle from exceeding total within a window, ensuring that the decomposition cannot create energy.

Architectural Consequences. The resulting per-domain $(\Delta E_{d,k}^{\text{idle}}, \Delta E_{d,k}^{\text{dyn}})$ split provides a stable baseline for policy-driven idle allocation while reserving fine-grained attribution capacity for the dynamic remainder. Energy conservation holds per window by construction, enabling downstream stages to distribute dynamic energy without ambiguity and to route any non-attributable portions explicitly to the system bucket.

4.8.4.2 Residual Idle and Dynamic Decomposition

Problem Statement. Residual energy is defined as the portion of system-level energy not explained by explicitly modeled components. While this residual budget is energy-consistent by construction, its instantaneous power signal is affected by temporal misalignment and measurement latency. As a result, residual power cannot be interpreted uniformly across analysis windows. A decomposition is required that preserves conservation of the residual budget while preventing transient artifacts from corrupting persistent state or downstream learning.

Conceptual Model. The residual budget is decomposed into two modeled components: a residual idle component and a residual dynamic component. Residual idle represents a persistent, low-variance baseline within the residual budget, while residual dynamic captures the remaining window-local variation. The decomposition is explicitly conditional. Its semantic validity depends on a window usability predicate that identifies windows dominated by temporal misalignment. When this predicate is false, the decomposition remains defined but must not be interpreted or used for learning.

Formalization. Let P_w^{res} denote the non-negative residual total power for window w. For every window satisfying the usability predicate, the residual decomposition is defined as:

$$P_w^{\text{res}} = P_w^{\text{res,idle}} + P_w^{\text{res,dyn}} \tag{4.17}$$

with the invariants:

$$P_w^{\text{res,idle}} \geq 0, \qquad P_w^{\text{res,dyn}} \geq 0, \qquad P_w^{\text{res,idle}} \leq P_w^{\text{res}} \tag{4.18}$$

Residual idle power is defined using a persistent modeled baseline B^{res}:

$$P_w^{\text{res,idle}} = \min\left(B^{\text{res}}, P_w^{\text{res}}\right) \tag{4.19}$$

Residual dynamic power is defined by conservation:

$$P_w^{\text{res,dyn}} = P_w^{\text{res}} - P_w^{\text{res,idle}} \tag{4.20}$$

The interpretive contract of this decomposition holds only when the window usability predicate is true. No fallback semantics are defined when this condition is violated.

Design Decisions. Residual idle is modeled as a persistent baseline rather than a per-window estimate. This choice prevents transient measurement artifacts from collapsing the baseline and destabilizing downstream stages. The model deliberately avoids any physical interpretation of residual idle or dynamic components. By enforcing exact conservation and non-negativity, the decomposition remains bounded and conservative under all conditions.

Architectural Consequences. This decomposition provides a stable internal structure over unmodeled energy while explicitly limiting its semantic scope. Downstream stages may rely on residual idle and dynamic quantities only when window usability is asserted. At the same time, the model constrains interpretation by requiring consumers to treat usability as a hard semantic gate rather than a soft quality indicator.

4.8.4.3 GPU Idle and Dynamic Decomposition

Problem Statement. The corrected GPU power signal contains both an idle baseline and workload-induced dynamic consumption. For attribution and energy accounting, Tycho must separate these components such that idle reflects the lowest sustainably attainable device power under low utilization, while dynamic captures the residual induced by activity. This separation must be defined per device, since baseline power differs across GPUs and operating conditions.

Conceptual Model. For each GPU device identified by *uuid*, Tycho maintains an idle power estimate β_{uuid} derived from the corrected total power signal under low and stable utilization. The dynamic power is defined as the non-negative residual between total and idle. Idle and dynamic energy increments are obtained by applying the same window duration used for the corresponding total power observation, ensuring that the decomposition is temporally consistent within each window.

Formalization. Let $P_{uuid}(w)$ denote the corrected total GPU power associated with analysis window w, and let $\beta_{uuid}(w)$ denote the idle power estimate available for

that window. Tycho defines the power decomposition as

$$P^{\text{idle}}_{uuid}(w) = \text{clip}(\beta_{uuid}(w),\, 0,\, P_{uuid}(w))\,, \tag{4.21}$$

$$P^{\text{dyn}}_{uuid}(w) = \max\Big(0,\, P_{uuid}(w) - P^{\text{idle}}_{uuid}(w)\Big)\,. \tag{4.22}$$

With window duration $\Delta t(w)$, the corresponding per-window energy increments are

$$\Delta E^{\text{idle}}_{uuid}(w) = P^{\text{idle}}_{uuid}(w)\,\Delta t(w), \tag{4.23}$$

$$\Delta E^{\text{dyn}}_{uuid}(w) = P^{\text{dyn}}_{uuid}(w)\,\Delta t(w). \tag{4.24}$$

The decomposition is conservative by construction:

$$\Delta E^{\text{idle}}_{uuid}(w) + \Delta E^{\text{dyn}}_{uuid}(w) = P_{uuid}(w)\,\Delta t(w), \tag{4.25}$$

up to discretization and clipping effects, and with all terms constrained to be non-negative.

Design Decisions. Tycho derives β_{uuid} from the corrected power signal rather than raw device telemetry to preserve temporal consistency with the exported total power and to avoid bias from known sampling and delay artifacts. Idle estimation is gated by stability of recent utilization observations and restricted to low-utilization regimes to prevent transient load onsets from being absorbed into the baseline. The model is defined per device to avoid cross-GPU coupling and to allow heterogeneous baselines within the same node.

Architectural Consequences. Downstream stages may treat $(P^{\text{idle}}_{uuid}, P^{\text{dyn}}_{uuid})$ and their cumulative energies as a stable, non-negative decomposition of corrected total power per device. The stability gate implies that β_{uuid} evolves only under sufficiently stationary conditions, so dynamic power absorbs short-lived transients by default. This design intentionally prioritizes robustness of the idle baseline over rapid adaptation to changing operating points.

4.8.5 Stage 4: Workload Attribution and Aggregation

Stage 4 realises workload-level attribution by mapping system- and component-level signals to concrete execution contexts. It defines the admissible scope of attribution, resolves workload identity under partial observability, and materialises utilization metrics and CPU and GPU energy at the workload level, while explicitly bounding what is not attributed.

4.8.5.1 Workload Resolution and the `__system__` Class

Problem Statement. Energy signals observed by Tycho are not intrinsically associated with Kubernetes workloads. Attribution therefore requires an explicit resolution step that maps low-level execution identities to container, pod, and namespace labels. This mapping is inherently partial, asynchronous, and failure-prone due to process churn, PID reuse, delayed metadata visibility, and non-Kubernetes activity. An architectural treatment is required to ensure that attribution remains total, conservative, and interpretable under these conditions.

Conceptual Model. Stage 4 operates on energy quantities that are already temporally aligned and semantically classified (total, idle, dynamic). Workload attribution is defined as a conditional labeling step that associates each attributable energy portion with a Kubernetes workload identity when such an identity can be resolved with sufficient confidence. Resolution is based on a best-effort join between execution-level observations and a metadata view that reflects the current Kubernetes state. Energy portions for which no workload identity can be resolved are assigned to a distinguished workload class, denoted `__system__`, which represents non-attributable, infrastructural, or unresolved activity.

Formalization and Invariants. Let E denote an energy quantity produced by earlier stages and let $\mathcal{W}$ be the set of resolvable Kubernetes workload identities. Workload attribution defines a partial mapping $f : E \to \mathcal{W}$. To preserve totality, this mapping is extended to a total function $\hat{f} : E \to \mathcal{W} \cup \{\texttt{__system__}\}$ by assigning all unresolved energy to `__system__`. The following invariants hold for all Stage 4 outputs: (i) conservation: the sum of workload-attributed energy equals the input energy, (ii) non-negativity: no workload receives negative energy, (iii) monotone degradation: loss of metadata resolution increases the share attributed to `__system__` but never reallocates energy between resolved workloads.

Design Decisions. The `__system__` label is treated as a first-class workload class rather than an absence of labeling. This avoids silent energy loss, preserves comparability across runs with different metadata quality, and makes unresolved attribution explicit. Workload resolution is deliberately decoupled from the attribution mathematics itself: the attribution model consumes resolved identities as inputs but does not embed resolution logic into its equations. This separation ensures that attribution correctness degrades conservatively under partial observability.

Architectural Consequences. All workload-attributed metrics in Stage 4 share a common resolution boundary and a uniform interpretation of unattributable energy. Metric-specific attribution logic may differ in how energy is distributed among resolved workloads, but unresolved energy is always surfaced explicitly via `__system__`. This abstraction enables consistent downstream analysis while preventing speculative attribution beyond what the available metadata supports.

4.8.5.2 Attribution Goals, Admissible Degrees of Freedom, and Temporal Granularity

Problem Statement. Even when workload identities are available, distributing energy among workloads is fundamentally underdetermined. Energy signals originate from shared components, while explanatory signals (utilization, activity, requests) are incomplete, indirect, and metric-specific. An architectural framing is required that defines what attribution must guarantee, what it may choose freely, and where Tycho deliberately refrains from speculative precision.

Attribution Goals and Invariants. Stage 4 attribution is governed by a small set of global goals that apply uniformly across metrics. Attribution must conserve energy, remain non-negative, and be complete over the workload domain extended by `__system__`. Attribution must not retroactively reinterpret past decisions and must degrade monotonically under loss of explanatory signals. Beyond these invariants,

attribution is explicitly not required to recover a unique or physically exact decomposition.

Admissible Degrees of Freedom. Within these constraints, Tycho allows metric-specific definitions of *fairness*. A metric may distribute energy proportionally to observed activity, declared resource requests, or other admissible proxies, provided that the global invariants are preserved. Fallback rules are permitted when proxies are missing or ill-defined, but such fallbacks must be explicit and must not introduce hidden redistribution between resolved workloads. This separation allows each metric to encode its own fairness assumptions without contaminating the shared attribution framework.

Fine-Grained Temporal Attribution as a Core Principle. A defining architectural choice of Tycho is that attribution is performed at the finest temporal granularity supported by the underlying signal, prior to any window-level aggregation. Energy portions are first attributed to workloads at this fine granularity and only then aggregated over the analysis window. This design enables Tycho to distinguish workloads with different energy characteristics even when their aggregate utilization within a window is similar. In contrast to window-pooled attribution schemes, this approach preserves temporal structure as an explanatory signal and avoids conflating heterogeneous workload behavior.

Architectural Consequences. Stage 4 defines attribution as a constrained optimization problem with explicit degrees of freedom rather than a fixed formula. Metric-specific attribution sections instantiate these freedoms while inheriting the same invariants and temporal discipline. As a result, Tycho can express nuanced workload energy differentiation when supported by data, while remaining conservative and interpretable when it is not.

4.8.5.3 Explicit Non-Attribution of Redfish Metrics

Redfish provides system-level power measurements that lack reliable workload-level explanatory signals. Attributing these measurements to Kubernetes workloads would therefore require speculative assumptions about the contribution of individual activities to shared components such as memory subsystems, interconnects, storage, or firmware-controlled behavior. Such assumptions are incompatible with Tycho's accuracy-first design philosophy.

Consequently, Redfish metrics are excluded from Stage 4 workload attribution. They are neither distributed across workloads nor labeled with `__system__` identities. Instead, Redfish metrics remain system-scoped signals that may be used for validation, bounding, or cross-checking of attributed energy, but not as sources of workload-resolved energy data.

This exclusion is a deliberate architectural decision rather than a limitation of the implementation. It preserves semantic clarity of workload-attributed metrics and prevents the introduction of unverifiable attribution artifacts.

4.8.5.4 Workload Attribution of eBPF Utilization Counters

Problem Statement. Fine-grained execution activity signals collected via eBPF are emitted at process scope and are not intrinsically associated with Kubernetes workloads. To support workload-level analysis, these signals must be attributed to workload identities without modeling assumptions, while preserving conservation, non-negativity, and monotone degradation under partial observability. Unlike energy signals, eBPF counters represent directly observed activity and therefore admit attribution by aggregation rather than distribution, but only if unresolved activity is handled explicitly and conservatively.

Conceptual Model. Each eBPF utilization signal is treated as a stream of per-process activity deltas observed at high temporal resolution. Stage 4 attribution maps each process delta to a workload identity when resolution succeeds, and otherwise assigns it to a distinguished `__system__` workload class. Workload-level utilization is obtained by summing all resolved process deltas per workload, while `__system__` captures the residual activity not attributable to any resolved workload. No reweighting, normalization, or inference across workloads is performed.

Formalization. Let $\Delta u_p(t)$ denote the utilization delta of process p over an attribution interval at time t, and let $R(p,t)$ be the workload resolution function, which yields either a workload w or $\varnothing$.

The workload-attributed utilization $U_w(t)$ is defined as

$$U_w(t) = \sum_{p|R(p,t)=w} \Delta u_p(t), \tag{4.26}$$

and the system residual is defined by construction as

$$U_{\texttt{\_\_system\_\_}}(t) = \sum_p \Delta u_p(t) - \sum_{w \neq \texttt{\_\_system\_\_}} U_w(t). \tag{4.27}$$

This definition guarantees conservation, completeness, and non-negativity, independent of resolution success.

Design Decisions. Utilization attribution is realized as aggregation rather than proportional distribution, reflecting that eBPF counters already encode directly attributable activity. The `__system__` workload is defined as a residual class by construction rather than as a fallback resolution outcome, ensuring that all unresolvable activity is captured without bias toward resolved workloads. Alternative schemes that discard unresolved activity or redistribute it among workloads were rejected due to violation of conservation and monotone degradation requirements.

Architectural Consequences. The resulting workload-level utilization counters form a complete and conservative attribution surface for execution activity. They are directly comparable across workloads and over time, and remain interpretable under partial observability. These metrics provide the explanatory substrate required for energy attribution analysis and validation, without introducing additional modeling assumptions or temporal coupling.

4.8.5.5 CPU Dynamic Energy Attribution

Problem Statement. CPU dynamic energy is exposed only as an aggregate per-domain signal for each RAPL CPU domain (`pkg`, `core`, `uncore`, `dram`) and is not intrinsically associated with individual workloads. At the same time, workload activity is observable only indirectly through execution-level proxies with finite resolution and partial coverage. The core problem is therefore to distribute a domain-level dynamic energy budget across workloads in a way that is conservative, temporally faithful, and robust to incomplete observability, while ensuring that attribution degrades monotonically toward the `__system__` class rather than redistributing energy among resolved workloads.

Conceptual Model. For a given analysis window W and RAPL domain B, the authoritative input is the window-level dynamic energy budget $\Delta E_{B,\mathrm{dyn}}(W)$. Attribution proceeds at native eBPF bin resolution and follows a two-stage model. First, the window-level energy budget is distributed across fine-grained temporal bins proportionally to observed CPU activity in each bin. Second, each bin's energy share is distributed across workloads proportionally to their activity within that bin. This preserves temporal structure as an explanatory signal while ensuring that attribution decisions are local to each bin and degrade conservatively when activity proxies are absent.

Formalization. Let $b \in W$ index the 50 ms bins contained in window W. For a given domain B, let $T_B(b)$ denote the total activity mass observed in bin b, and let $W_B(\ell, b)$ denote the activity mass attributed to workload ℓ in bin b. The per-bin energy share is defined as

$$\Delta E_B(b) = \begin{cases} \Delta E_{B,\mathrm{dyn}}(W) \cdot \dfrac{T_B(b)}{\sum_{b' \in W} T_B(b')} & \text{if } \sum_{b' \in W} T_B(b') > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{4.28}$$

Within each bin, workload attribution is defined as

$$\Delta E_B(\ell, b) = \begin{cases} \Delta E_B(b) \cdot \dfrac{W_B(\ell, b)}{T_B(b)} & \text{if } T_B(b) > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{4.29}$$

If either $\sum_b T_B(b) = 0$ or $T_B(b) = 0$ for a given bin, the corresponding energy mass is assigned to the `__system__` workload, preserving completeness and conservation.

Design Decisions. Observed CPU activity is used as the sole fairness basis. For the `pkg`, `core`, and `uncore` domains, activity is measured via per-process retired instruction counts. For the `dram` domain, activity is measured via cache-miss counts, with a window-scoped fallback to CPU cycle counts when cache-miss mass is zero but cycle mass is non-zero. This choice reflects the closest available causal relationship between workload behavior and domain-specific dynamic energy, while remaining explicit and conservative under missing signals.

Architectural Consequences. The resulting attribution is causal, temporally fine-grained, and strictly conservative. All dynamic CPU energy is assigned either to resolved workloads or to `__system__`, with no redistribution between workloads

under metadata loss. The bin-level structure remains an explicit part of the architectural contract, enabling later extensions such as non-linear attribution rules or idle–dynamic decomposition without revisiting the core attribution model.

4.8.5.6 CPU Idle Energy Attribution

Problem Statement. CPU idle energy represents background consumption that cannot be causally attributed to instantaneous workload execution. Unlike dynamic CPU energy, idle energy persists in the absence of activity and reflects platform state, reservation, and scheduling slack. Attribution must therefore define a fairness policy that preserves strict conservation and monotone degradation under partial observability, without asserting unsupported causal relationships.

Conceptual Model. Idle energy attribution is modeled as a policy-driven distribution of a per-window idle energy budget across workloads. The model separates declarative reservation from observed activity by partitioning the idle budget into two conceptual pools. Reservation captures long-lived entitlement to capacity, while activity captures short-term presence and participation. The distinguished `__system__` workload is a first-class participant that absorbs unavoidable background and remainder energy.

Formalization. For each analysis window W and RAPL domain d, a conservative idle energy budget $E^{d}_{\text{idle}}(W)$ is given. A reservation fraction $\beta \in [0, 1]$ partitions this budget:

$$E^{d,\text{req}}_{\text{idle}}(W) = \beta \cdot E^{d}_{\text{idle}}(W), \qquad E^{d,\text{opp}}_{\text{idle}}(W) = E^{d}_{\text{idle}}(W) - E^{d,\text{req}}_{\text{idle}}(W). \tag{4.30}$$

For CPU domains (`pkg`, `core`, `uncore`), β is derived from the ratio of declared CPU requests to node CPU capacity. For `dram`, β is derived analogously from declared memory requests and node memory capacity.

The reserved pool $E^{d,\text{req}}_{\text{idle}}$ is distributed proportionally among workloads with strictly positive requests. The opportunistic pool $E^{d,\text{opp}}_{\text{idle}}$ is distributed proportionally according to window-aggregated dynamic activity weights. Workloads without requests participate only in the opportunistic pool. The `__system__` workload always participates in the opportunistic pool and receives any residual energy required for exact conservation.

Design Decisions. A two-pool policy is chosen over pure activity-based allocation to preserve the semantic distinction between reserved capacity and opportunistic use. Using conservative per-window idle budgets preserves strict decomposition of total CPU energy into idle and dynamic components. Applying the same attribution structure across all CPU-related RAPL domains ensures interpretability and comparability, while allowing domain-specific reservation signals.

Architectural Consequences. Idle energy attribution is explicitly defined as a fairness policy rather than a causal reconstruction. The model guarantees conservation, non-negativity, and completeness per window, and degrades monotonically toward `__system__` under reduced observability. The resulting workload-level idle energy

counters are directly comparable to dynamic CPU attribution outputs and provide a stable basis for downstream analysis without retroactive reinterpretation.

4.8.5.7 GPU Dynamic Energy Attribution

Problem Statement. GPU dynamic energy must be attributed to Kubernetes workloads without directly observable per-workload energy counters. Attribution must remain conservative under incomplete proxy coverage and must not force energy onto workloads when the proxy signal cannot support it.

Conceptual Model. For each GPU device, dynamic energy is treated as a window budget derived from the corrected GPU energy signal. Workloads receive shares of this budget proportional to per-process GPU compute utilization, aggregated over the window and mapped to workloads via the established resolver. Any fraction of dynamic energy that cannot be supported by resolvable process activity is assigned to `__system__`.

Formalization. Let W be an analysis window and let $\Delta E_{\text{gpu}}(W, \text{uuid})$ denote the corrected total GPU energy over W for device uuid. Let $\Delta E_{\text{gpu,dyn}}(W, \text{uuid})$ denote the dynamic GPU energy budget for the same window and device. Tycho allocates dynamic energy at the granularity of process-sample intervals. For an interval $I \subseteq W$, let $\Delta E_{\text{gpu}}(I, \text{uuid})$ be the corrected interval energy, and let

$$f(W, \text{uuid}) = \text{clip}_{[0,1]}\left(\frac{\Delta E_{\text{gpu,dyn}}(W, \text{uuid})}{\Delta E_{\text{gpu}}(W, \text{uuid})}\right). \tag{4.31}$$

The interval dynamic budget is

$$\Delta E_{\text{gpu,dyn}}(I, \text{uuid}) = f(W, \text{uuid}) \cdot \Delta E_{\text{gpu}}(I, \text{uuid}). \tag{4.32}$$

Let $\mathcal{P}_I$ be the set of observed GPU PIDs in I, and let $u_p(I) \in [0, 100]$ be the held utilization proxy for PID p over I. Define proxy mass

$$U(I) = \sum_{p \in \mathcal{P}_I} \max(0, u_p(I)). \tag{4.33}$$

If $U(I) = 0$, then $\Delta E_{\text{gpu,dyn}}(I, \text{uuid})$ is assigned to `__system__`. Otherwise, each PID receives a process-level share

$$\Delta E_{\text{gpu,dyn}}(p, I, \text{uuid}) = \Delta E_{\text{gpu,dyn}}(I, \text{uuid}) \cdot \frac{u_p(I)}{U(I)}. \tag{4.34}$$

Each process share is mapped to a workload key via the resolver; unresolved shares are assigned to `__system__`. Workload energy is the sum of mapped process shares across all intervals in W and all contributing PIDs.

Design Decisions. Dynamic GPU energy is distributed exclusively using per-process `ComputeUtil` as the proxy basis. This choice preserves conservative semantics: any dynamic energy not supported by positive proxy mass, or not attributable through the resolver, is routed to `__system__`. A per-window dynamic fraction $f(W, \text{uuid})$ scales interval energy to reconcile sub-window corrected energy structure with a

separately constructed dynamic budget, without introducing an independent dynamic model at sub-window granularity.

Architectural Consequences. Dynamic GPU workload attribution is proxy-limited. Even under purely Kubernetes-driven GPU load, a non-zero `__system__` dynamic share is admissible and expected when driver overhead, sampling noise, or incomplete per-process accounting prevents full proxy coverage. Attribution remains conservative because unresolved or weakly supported energy increases `__system__` share rather than being forced onto resolved workloads.

4.8.5.8 GPU Idle Energy Attribution

Problem Statement. GPU idle energy lacks a defensible workload-level proxy and must not be speculatively distributed across workloads.

Conceptual Model. GPU idle energy is treated as non-attributable and is assigned in full to the canonical `__system__` workload class, per GPU device.

Formalization. For each window W and device uuid, let $\Delta E_{\text{gpu,idle}}(W, \text{uuid})$ be the corrected idle GPU energy. Tycho defines workload-attributed idle energy as

$$\Delta E_{\text{gpu,idle}}(\texttt{\_\_system\_\_}, W, \text{uuid}) = \Delta E_{\text{gpu,idle}}(W, \text{uuid}), \tag{4.35}$$

and emits no non-`__system__` idle workload series.

Design Decisions. Idle attribution is intentionally degenerate. The only emitted idle workload series is `__system__`, enabling a complete workload accounting view over GPU energy without introducing speculative distribution.

Architectural Consequences. GPU idle energy is explicitly interpreted as infrastructural or non-attributable device cost. Completeness is achieved by construction, while preserving the accuracy-first constraint that idle energy is not redistributed to workloads.

4.8.6 Prometheus Exporter

Problem Statement. Tycho produces window-committed analysis points that must be exposed to external systems without introducing additional semantics, temporal reinterpretation, or ownership over metric meaning.

Conceptual Model. The exporter is defined as a passive sink that materializes the latest committed analysis points into a pull-based observation interface. It does not interpret, aggregate, or transform metrics beyond name normalization and schema stabilization. Exposition semantics, scraping cadence, retention, and downstream labeling are explicitly outside Tycho's architectural scope.

Formalization. For each analysis point emitted at the end of an analysis cycle, the exporter exposes exactly one observable value per metric key corresponding to the most recently committed window. No historical state, interpolation, or backfilling is performed.

Design Decisions. Metric exposition is deliberately decoupled from analysis execution. Tycho does not define scrape timing, persistence, or query semantics, and does not distinguish between metric consumers. All exported metric names are prefixed with `tycho_` to ensure namespace isolation and unambiguous attribution.

Architectural Consequences. Metric meaning and correctness are fully determined upstream of the exporter. Downstream systems may freely attach additional labels such as node identity or perform aggregation without affecting Tycho's attribution guarantees.

4.9 Architectural Trade-Offs and Alternatives Considered

Tycho adopts an explicitly *accuracy-first* architectural stance. This choice prioritises preservation of temporal structure and interpretability of uncertainty over simplicity of implementation. As a result, the system deliberately rejects several simpler architectural alternatives that would reduce complexity by discarding information or by enforcing assumptions that are incompatible with heterogeneous, asynchronous metric sources. This section briefly situates Tycho's design within the broader architectural design space and clarifies why these alternatives were not adopted.

4.9.1 Timing and Data Collection Models

A common design choice in monitoring systems is to impose a single global polling loop or a strictly synchronised sampling schedule across all metric sources. Such designs are attractive because they simplify implementation and produce visually aligned time series. However, this alignment is artificial: it reflects the sampling strategy rather than the behaviour of the underlying systems.

Tycho rejects globally synchronised sampling because the metric domains it observes do not share a natural clock. CPU energy counters, accelerator telemetry, kernel execution events, and out-of-band system power measurements each exhibit distinct update semantics and latency characteristics. Forcing these domains into a single synchronous schedule necessarily introduces aliasing, under-sampling, or false simultaneity, thereby destroying temporal meaning rather than revealing it.

In an accuracy-first architecture, temporal structure is part of the signal. Tycho therefore treats domain-specific timing behaviour as a first-class property and preserves it through independent, domain-aware collection. The resulting increase in architectural complexity is a direct consequence of respecting how information is produced, rather than a by-product of implementation choices.

4.9.2 Attribution Model Design Space

Workload-level energy attribution is fundamentally underdetermined. Multiple attribution paradigms exist that reduce this ambiguity by imposing strong simplifying assumptions, such as static proportionality, fixed analytical models, or globally normalised distributions. While these approaches yield concise formulations, they also collapse uncertainty and obscure the limits of what can be inferred from the available data.

Tycho deliberately avoids committing to such simplifications at the architectural level. An accuracy-first system must remain explicit about what is supported by observation and what is not. This requires attribution mechanisms that degrade conservatively under partial observability, preserve conservation and non-negativity, and avoid redistributing energy based on unverifiable assumptions.

Accordingly, Tycho's architecture constrains attribution through invariants and admissible degrees of freedom rather than prescribing a single closed-form solution. This preserves interpretability and makes uncertainty visible, at the cost of increased conceptual and structural complexity.

4.9.3 Accuracy Versus Architectural Complexity

The architectural complexity of Tycho is not incidental. It arises from the decision to preserve information across heterogeneous domains, to maintain temporal coherence under asynchronous observation, and to expose uncertainty rather than masking it. Simpler architectures achieve tractability primarily by discarding these properties.

From an accuracy-first perspective, complexity is acceptable where it prevents semantic loss and bounded where it does not. Tycho confines complexity through explicit abstraction boundaries between collection, timing, analysis, and attribution, ensuring that individual components remain understandable even as the system as a whole addresses a challenging problem space.

Reducing architectural complexity in this context would primarily eliminate information and weaken interpretability rather than improve correctness. The resulting system might appear simpler, but it would do so by obscuring the very phenomena that Tycho is designed to measure.

4.10 Summary

This chapter defined the architectural foundation of Tycho as an accuracy-first energy attribution system for Kubernetes environments. The design is derived directly from the requirements established in Chapter 3, in particular temporal coherence across heterogeneous metric sources, domain-level consistency, transparent modelling assumptions, lifecycle-robust workload identity handling, and explicit treatment of uncertainty.

The architecture is organised around a small number of clearly separated concerns. Independent, domain-aware metric collectors acquire observations without imposing artificial synchrony. A global event-time-based timing engine defines attribution windows and provides a coherent temporal reference for all analysis. Calibration serves as an auxiliary subsystem that constrains polling and delay uncertainty where hardware-controlled publication behaviour would otherwise undermine temporal interpretation.

Analysis is structured as a deterministic, staged pipeline that transforms buffered observations into window-scoped quantities under explicit dependency and conservation rules. Component-level metric construction establishes authoritative energy and power signals, which are subsequently decomposed into idle and dynamic

contributions before being mapped to workload identities. Workload attribution is explicitly bounded: energy is attributed conservatively where explanatory signals permit, and unresolved or infrastructural consumption is surfaced explicitly via the `__system__` class rather than being redistributed speculatively.

Metadata collection and lifecycle management are treated as architectural prerequisites for attribution correctness rather than as auxiliary implementation details. By enforcing freshness bounds and separating identity resolution from attribution mathematics, the architecture ensures monotone degradation under partial observability. Finally, metric export is defined as a passive sink, preserving the semantic boundary between attribution logic and downstream monitoring systems.

Together, these architectural elements specify *what* Tycho must do and *why* it must do so to meet its accuracy and transparency goals, without committing to specific implementation mechanisms. The following chapter describes the concrete implementation of these architectural components, including the collectors, timing engine, metadata subsystem, calibration routines, and the realisation of the analysis and attribution pipeline.

Chapter 5

Implementation

5.1 Purpose, Scope, and Execution-Time Structure

This chapter explains how Tycho's architectural abstractions are realised at runtime under the constraints of discretization, partial observability, and asynchronous execution. Its role is to describe how responsibility boundaries defined in the architecture are enforced concretely, and how correct attribution is achieved despite imperfect and delayed inputs. Architectural concepts, models, and invariants are assumed from earlier chapters and are not reintroduced here.

At execution time, Tycho is structured as a set of long-lived subsystems with strictly separated responsibilities and unidirectional interaction. Each subsystem exercises authority over a narrow concern, and no subsystem compensates implicitly for the behaviour of others. This execution-time separation forms the foundation for correctness, auditability, and robustness throughout the implementation.

Additional implementation detail that is not required for understanding the runtime structure or correctness arguments is intentionally deferred to Appendix C. The appendix collects auxiliary material such as extended configuration descriptions, supporting scripts, and low-level operational notes that aid reproducibility and inspection without obscuring the main implementation narrative.

5.1.1 Runtime Subsystems and Responsibilities

Tycho's runtime consists of the following subsystems, each of which is examined in detail later in this chapter:

- The **timing engine** provides execution-time coordination by triggering collection and analysis actions according to a global schedule, without participating in interpretation or attribution (§ 5.2).
- **Metric collectors** act as independent observers that acquire raw measurements from individual hardware and software domains and emit timestamped samples without coordination or semantic interpretation (§ 5.4).
- The **metadata subsystem** maintains a refreshed view of workload identity and hierarchy, supplying identity context during attribution without joining metric streams or performing analysis (§ 5.5).

- **Calibration mechanisms** derive auxiliary parameters that characterise source behaviour and contextualise interpretation, executing outside steady-state attribution and without modifying observations (§ 5.6).
- The **analysis engine** is the sole authority responsible for interpreting observations, fusing domains, applying attribution models, and enforcing architectural invariants on a per-window basis.
- **Export** observes the results of analysis and exposes them to external systems without influencing upstream execution or attribution semantics

5.1.2 Execution-Time Interaction Model

Interaction between these subsystems follows a strictly unidirectional pattern. Temporal authority originates in the timing engine, observation authority in collectors and metadata acquisition, and semantic authority exclusively in the analysis engine. Data flows forward through explicit handoff only: raw observations and identity context are materialised upstream and consumed read-only during analysis, while attribution results flow downstream to export.

This interaction model deliberately excludes feedback paths, implicit coordination, and retroactive modification of observations. Once emitted, samples are immutable; once a window is analysed, its results are final. These constraints ensure that attribution semantics remain explicit, reproducible, and independent of scheduling or export behaviour.

The remainder of this chapter elaborates on how each subsystem realises its assigned responsibility in practice, addressing temporal realisation, collection mechanics, identity handling, calibration, attribution, and robustness in turn (§ 5.2–§ **??**).

5.2 Temporal Infrastructure and Window Realization

5.2.1 Architectural Context and Implementation Problem

§ 4.4 defines Tycho's temporal model in abstract terms: a single monotonic time base, independently operating collectors, and fixed-duration analysis windows triggered by a timing engine. The implementation task is to realize this model under real execution constraints while preserving its guarantees, rather than restating its semantics.

Collectors are scheduled by a general-purpose operating system and are subject to jitter, preemption and variable execution latency. Polling callbacks may execute late, at uneven intervals, or out of phase with one another. Analysis must therefore not depend on execution order, callback timing, or implicit synchronization effects. Temporal correctness must derive exclusively from explicit timestamps attached to observations, not from when code happens to execute. The temporal infrastructure enforces this separation rigorously.

5.2.2 Global Monotonic Time Realization

The architectural event-time model relies on a single system-wide monotonic time base. Its implementation elevates monotonic time to a first-class dependency rather

than treating it as an incidental property of the runtime environment. All collectors, the timing engine and the analysis engine obtain temporal information exclusively through a dedicated clock abstraction.

Wall-clock time is excluded from analysis-critical paths and appears only where external representation is unavoidable. The clock abstraction provides monotonic timestamps for observations and analysis boundaries, mediates conversion between real-time and monotonic representations where required, and is injected into downstream components to ensure consistent and testable temporal behavior across subsystems.

As a result, scheduling jitter becomes an explicit input to analysis rather than a hidden source of error. A collector that executes late produces a correspondingly late timestamp. No corrective action is taken at collection time; timestamps are interpreted during analysis according to the delay and freshness assumptions defined in § 4.4. Monotonic timestamps thus constitute the sole temporal authority within the system.

5.2.3 Timing Engine and Hierarchical Cadence Alignment

Independent collector schedules are a central architectural principle. Realizing this independence without sacrificing determinism requires a controlled mechanism for initiating periodic actions. Tycho employs a centralized timing engine to which all periodic activities register during system initialization.

Each registration specifies a period expressed as an integer multiple of a global base quantum (default: 1 ms). All registrations are aligned to a shared epoch defined by this quantum, establishing deterministic phasing across the system. Collector and analysis triggers are hierarchically derived from this common cadence rather than started opportunistically, ensuring that identical configurations produce identical temporal behavior across runs. Alignment does not impose a shared frequency: collectors with different periods remain independent, but their triggers occur at deterministic offsets on the global monotonic axis.

The timing engine is deliberately non-semantic. It does not inspect collected data, adapt schedules, or coordinate collectors. Its sole responsibility is to emit triggers at predetermined monotonic times. Work performed in response to a trigger is constrained to be minimal and non-blocking, typically limited to recording an observation and placing it into a buffer, preventing local execution delays from propagating into global timing behavior.

Analysis triggering is implemented using the same registration mechanism. The analysis engine registers a periodic trigger alongside collectors, making analysis execution subject to the same alignment and determinism guarantees. This design enforces single-cycle exclusivity by construction: a new analysis cycle cannot begin before the previous trigger boundary has been established, without requiring additional synchronization logic.

5.2.4 Analysis Window Realization and Trigger Semantics

Analysis windows are realized when an analysis trigger fires. At that instant, the timing engine provides a single monotonic timestamp t_{now}, which defines the upper boundary of the current window. This timestamp is captured exactly once and propagated unchanged throughout the entire analysis cycle. All metrics are evaluated against windows derived from this shared boundary, and no component recomputes or refines the window definition during analysis. Consequently, all attribution decisions within a cycle refer to an identical temporal interval, independent of execution order or internal processing latency.

Window boundaries are defined by trigger times rather than sample arrival. Samples collected before t_{now} may be included or excluded according to domain-specific delay and freshness rules as defined in § 4.4. Samples arriving after the trigger are attributed to subsequent windows. This separation prevents double-counting and omission even under heterogeneous collector rates.

Temporal complexity is intentionally confined to timestamping and delay interpretation. Window construction itself remains simple and predictable, providing a stable temporal substrate on which later analysis stages can reason about delay, partial observation and attribution correctness without embedding scheduling assumptions or compensating for execution artifacts.

5.3 Historical Observation Retention

Tycho retains a bounded history of raw observations in order to support downstream analysis that requires temporal context beyond a single attribution window. This retention is an explicit implementation responsibility derived from the temporal model in § 4.4. Rather than operating exclusively on window-local samples, Tycho preserves historical signal to mitigate discretization effects, tolerate heterogeneous collector cadences, and enable mathematically stable downstream interpretation.

5.3.0.1 Metric Observation Retention

Historical retention is realized through per-collector observation buffers with time-based semantics. Each collector appends observations to its own buffer, which retains a fixed-duration history with a default horizon of approximately 90 s. This horizon deliberately exceeds the nominal analysis window length and is chosen to provide substantial temporal context for downstream analysis. Buffer capacity is computed at startup from the collector's polling interval and the configured analysis window, ensuring that retained history covers at least twice the longer of these durations, augmented by a small safety margin. Under Tycho's default configuration for high-frequency analysis, this corresponds to retention spanning approximately 18 full analysis windows, providing substantial historical context for downstream analysis models. Retention is bounded and fixed for the lifetime of the process.

Buffered samples are append-only and immutable once written. Downstream components access buffered data strictly in a read-only manner. No guarantee is made

that all collectors contribute samples to every window or that samples are temporally aligned across collectors. Partial observability and heterogeneous update patterns are therefore preserved explicitly and interpreted by the analysis engine rather than hidden by synchronization.

5.3.0.2 Metadata Retention

In addition to metric observations, Tycho maintains a bounded cache of metadata describing process, cgroup and workload identities. This cache employs a time-based retention policy aligned with the metric retention horizon to ensure that buffered observations can be joined with valid identity information during analysis. Metadata history is not used for long-term modeling and is removed once it exceeds the retention window.

5.4 Metric Collection Subsystems

5.4.1 eBPF Collector Implementation

This section describes how Tycho realises the event-driven CPU ownership and activity model introduced in § 4.5.1. The eBPF collector implements a kernel-level acquisition path that captures execution boundaries precisely and exposes the resulting activity as bounded, per-window deltas for downstream analysis.

5.4.1.1 Implementation Strategy

The implementation follows a split-surface strategy that separates *attributable* activity from *non-attributable* CPU time. Attributable activity is accumulated per process at scheduler boundaries, together with stable identity and classification metadata. Non-attributable activity, including idle time and interrupt handling, is accumulated per CPU and exported independently. This separation reflects Tycho's attribution requirements: process-level ownership must be preserved without ambiguity, while certain CPU time categories cannot be meaningfully assigned to workloads. All kernel programs operate strictly locally, without cross-CPU or cross-process aggregation; consolidation and interpretation are deferred to userspace and later analysis stages.

5.4.1.2 Core Mechanisms

Process-level accounting is driven by scheduler transitions. At each context switch, the outgoing execution interval is closed and its duration is accumulated into the corresponding process aggregate, together with hardware performance counters sampled at the same boundary. Process identity, control-group association, and kernel-thread classification are captured at these execution boundaries and stored alongside the counters. By aligning accumulation with actual ownership changes, the implementation preserves the architectural guarantee that execution intervals form precise attribution boundaries.

CPU-local accounting handles activity that is not attributable to individual processes. Each CPU maintains a local state machine that tracks the currently active context and the timestamp of the last transition. Idle time is detected explicitly via the scheduler's idle task and accumulated when the CPU executes in this state. Hard interrupt and soft interrupt handling are measured as outermost intervals using entry

and exit hooks, with durations accumulated into per-CPU bins. This design avoids double counting under nested interrupts while preserving a complete partition of CPU time at the node level.

Userspace collection materialises kernel-side accumulation into analysis-ready deltas. At a fixed polling cadence, the collector snapshots all process aggregates and resets only their counter fields while preserving identity metadata. This stable-key snapshot design avoids missing-entry artefacts that can occur if keys are deleted while scheduler updates are in flight. CPU-level bins are read and reset in the same cycle, defining a clear collection boundary for idle and interrupt time. The result of each collection cycle is a single tick record that represents all observed activity since the previous boundary, without overlap or double counting.

5.4.1.3 Robustness and Edge Cases

Several implementation choices ensure robustness under concurrent kernel activity. Process aggregates persist across collection cycles, and only delta-relevant fields are reset, preventing transient key loss under concurrent scheduler updates. All kernel-side state is bounded through per-CPU arrays, bounded per-process maps, and fixed-size histograms for interrupt vector enrichment. Reset failures or map evictions are treated as non-fatal; correctness is restored automatically in subsequent cycles. A fundamental observability limit remains: processes that execute entirely between two collection boundaries may not be observed. This limitation is inherent to discrete materialisation and is not compensated by inference.

5.4.1.4 Implementation Consequences

In practice, the eBPF collector produces a fixed-resolution utilisation and activity surface that preserves execution-boundary accuracy and stable process identity. Per-window deltas are exported without imposing additional timing constraints on the analysis engine, enabling proportional attribution and energy modelling in later stages.

5.4.1.5 Collected Metrics

The process-level and CPU-level metrics exported by the eBPF collector are listed in table Table 5.1.

Metric	Source hook	Description
Time-based metrics		
Process runtime	`tp_btf/sched_switch`	Per process. Elapsed on-CPU time accumulated at context switches.
Idle time	Derived from `sched_switch`	Per node. Aggregated idle time across CPUs.
IRQ time	`irq_handler_{entry,exit}`	Per node. Aggregated duration spent in hardware interrupt handlers.
SoftIRQ time	`softirq_{entry,exit}`	Per node. Aggregated duration spent in deferred kernel work.
Hardware-based metrics		
CPU cycles	PMU (`perf_event_array`)	Per process. Retired CPU cycle count during task execution.
Instructions	PMU (`perf_event_array`)	Per process. Retired instruction count.
Cache misses	PMU (`perf_event_array`)	Per process. Last-level cache misses; indicator of memory intensity.
Classification and enrichment metrics		
Cgroup ID	`sched_switch`	Per process. Control group identifier for container attribution.
Kernel thread flag	`sched_switch`	Per process. Marks kernel threads executing in system context.
Page cache hits	`mark_page_accessed`	Per process. Read or write access to cached pages; proxy for I/O activity.
IRQ vectors	`softirq_entry`	Per process. Frequency of specific soft interrupt vectors.

TABLE 5.1: Metrics collected by the kernel `eBPF` subsystem.

5.4.2 RAPL Collector Implementation

This section realizes the architectural model of cumulative CPU-domain energy sampling described in § 4.5.2. The collector's responsibility is strictly limited to observing hardware-provided cumulative energy counters at tick boundaries and preserving their semantics.

5.4.2.1 Implementation Strategy

To preserve domain-consistent CPU energy measurement across vendors, the collector employs a dual-backend strategy selected at runtime via CPUID. On Intel systems, energy is obtained through the RAPL interface exposed via the `powercap` subsystem. On AMD systems, the collector preferentially uses `amd_energy` via the `hwmon` interface, which provides accurate CPU energy telemetry on these platforms. The powercap path is used on AMD only if no CPU-labeled hwmon energy source is present. This strategy ensures that the architectural RAPL domain abstraction is realized consistently, independent of vendor-specific exposure mechanisms.

5.4.2.2 Core Mechanisms

At each tick, the collector obtains a single snapshot of cumulative energy counters for all available CPU domains and sockets and records them unchanged. All values

are stored as cumulative microjoule counters, matching the native units of the underlying sources. Supported domains include package and core on all platforms, with uncore (PP1) and DRAM recorded when exposed by the hardware. On AMD systems, `amd_energy` publishes per-logical-core energy values; these are aggregated by summation into a single cumulative core-domain counter to preserve domain semantics. Domains not provided by the platform are recorded as zero to maintain a stable domain set across ticks.

5.4.2.3 Robustness and Edge Cases

Powercap-backed RAPL counters may wrap and are corrected at collection time to maintain monotonicity. The hwmon-backed `amd_energy` counters do not wrap and require no correction. Each sample is tagged exclusively with a monotonic timestamp provided by Tycho's timing engine. Partial reads are permitted and result in partial ticks; no backend instability was observed in practice.

5.4.2.4 Implementation Consequences

In practice, the collector guarantees exactly one cumulative energy sample per supported domain and socket at each tick, with vendor-independent domain semantics and bounded noise. The resulting time series form a stable input for downstream differencing and attribution. The only remaining limitation is platform-dependent domain availability, which is intentionally exposed rather than approximated.

5.4.2.5 Collected Metrics

The RAPL collector exports raw cumulative energy counters once per tick. Table 5.2 summarizes the metrics recorded per `RaplTick`.

Metric	Unit	Description
Per-socket energy counters		
`Pkg`	mJ	Cumulative package energy per socket (RAPL `PKG` domain).
`Core`	mJ	Cumulative core energy per socket (RAPL `PP0` domain), when available.
`Uncore`	mJ	Cumulative uncore energy per socket (RAPL `PP1` or uncore domain), when available.
`DRAM`	mJ	Cumulative DRAM energy per socket (RAPL `DRAM` domain), if the platform exposes it.
Metadata		
`Source`	–	Identifier of the active RAPL backend (for example `powercap`)
`Sockets`	–	Map from socket identifier to the corresponding set of domain counters.
`SampleMeta.Mono`	–	Monotonic timestamp assigned by Tycho's timing engine at the moment of collection.

TABLE 5.2: Metrics exported by the RAPL collector per `RaplTick`.

5.4.3 Redfish Collector Implementation

This section describes how Tycho realizes the Redfish power source defined in § 4.5.3. Redfish is treated as an externally clocked, latently published observation whose update cadence and timing semantics are controlled by the Baseboard Management Controller. The implementation must therefore tolerate missing observations, expose temporal uncertainty explicitly, and avoid manufacturing samples while still enabling a coherent downstream power timeline.

5.4.3.1 Implementation Strategy

The Redfish collector issues at most one query per engine tick and emits a record only when a power value can be obtained reliably or when an explicit continuity fallback is required. The collector is permitted to emit no record for a tick. This choice reflects the architectural intent to avoid speculative continuity and to preserve the distinction between absence of information and persistence of state. Temporal alignment to Tycho's monotonic timebase is achieved by timestamping at collection time, without attempting synchronization or correction of BMC time.

5.4.3.2 Freshness Realization and Semantics

Freshness is realized as a best-effort quality annotation computed as the difference between the local monotonic collection time and the timestamp provided by the BMC, when available. Given the limited and vendor-specific semantics of BMC timestamps, the collector applies no correction, filtering, or normalization. Freshness therefore represents observed latency and staleness rather than a validity constraint.

When continuation records are emitted, the collector reuses the most recent BMC timestamp and recomputes freshness accordingly. As a consequence, freshness increases during prolonged publication gaps, making temporal uncertainty explicit. The collector never suppresses, alters, or reclassifies samples based on freshness. All values are forwarded unchanged as downstream quality indicators.

5.4.3.3 Heartbeat-Based Continuity as Fallback

Irregular Redfish publication implies that fixed-cadence sampling may observe extended periods without new measurements. The collector therefore supports an optional heartbeat mechanism whose role is explicitly fallback-oriented. Heartbeat does not operate on every missed tick. Instead, it re-emits a specially marked continuation record only when no fresh observation has been obtained for a comparatively long interval relative to the engine cadence.

By default, this interval substantially exceeds the collection period, ensuring that short-lived access failures or transient gaps result in silence rather than artificial continuity. If enabled, the heartbeat threshold may be configured statically or derived adaptively from observed inter-arrival times of fresh Redfish updates, with conservative bounds to avoid pathological behavior under highly irregular BMC implementations. Heartbeat emission never invents new measurements. It explicitly signals persistence of the last known value when prolonged absence would otherwise break temporal continuity.

5.4.3.4 Robustness Under Partial Observation

Redfish access failures and missing timestamps are treated as normal operating conditions. If a Redfish query fails, the collector emits no record for that tick. No retries, backoff strategies, or suppression mechanisms influence the semantic output. Only when the heartbeat threshold is exceeded does the collector emit a continuation record, clearly distinguishing prolonged absence from transient failure.

Multiple chassis are handled independently. Freshness computation, heartbeat state, and continuation decisions are maintained per chassis and never synchronized across nodes. This preserves architectural assumptions about the independence of Redfish power sources in multi-node deployments.

5.4.3.5 Implementation Consequences

In practice, the collector emits at most one record per chassis per engine tick, with zero records as a valid and expected outcome. Continuity is preserved only when absence becomes prolonged, and even then without obscuring staleness. The resulting stream provides a stable, monotonic reference for total node power while exposing uncertainty rather than masking it.

This implementation anchors Tycho's global energy view and supports later reconciliation with in-band estimates without conflating observation authority or temporal semantics. Its limitations are deliberate. Temporal resolution and accuracy are bounded by the BMC, and no component-level attribution is attempted at this stage.

5.4.3.6 Collected Metrics

The Redfish collector emits instantaneous chassis power together with identity and temporal metadata. Only raw observations are produced. Derived quantities such as energy are computed by downstream analysis stages. The exported fields are summarized in Table 5.3.

Metric	Unit	Description
Primary power metric		
`PowerWatts`	W	Instantaneous chassis power reported by the BMC.
Temporal and identity metadata		
`ChassisID`	-	Identifier of the chassis or enclosure.
`Seq`	-	Server-provided sequence number indicating new measurements.
`SourceTime`	s	Timestamp provided by the BMC, if available.
`CollectorTime`	s	Local collection time of the measurement.
`FreshnessMs`	ms	Difference between `SourceTime` and `CollectorTime`.

TABLE 5.3: Metrics collected by the Redfish collector.

5.4.4 GPU Collector Implementation

The GPU collector realises the architecture described in § 4.5.4 and integrates accelerator telemetry into Tycho's unified temporal framework. In contrast to other energy domains, GPU telemetry is published at discrete, driver-controlled moments that are neither continuous nor externally observable. The implementation is therefore responsible for enforcing phase-aligned, event-driven sampling under partial observability, backend variability, and timing jitter, while preserving strict monotonic ordering across all domains.

The central implementation invariant is that at most one `GpuTick` is emitted per confirmed hardware publish, and that no tick is emitted without a detectable device update. All mechanisms described in this section exist to uphold this invariant in practice, including the integration of retrospective process-level telemetry under wall-clock semantics.

5.4.4.1 Implementation Strategy

The implementation treats GPU sampling as an inference problem rather than a periodic measurement task. Because the driver's publish cadence is implicit, polling is used only as a means to detect new hardware updates, not as a proxy for time. Sampling effort is modulated according to the phase-aware timing model defined in § 4.5.4, concentrating observation near predicted publish moments while suppressing redundant reads elsewhere.

Freshness detection and event emission are deliberately decoupled. Polling may occur at high frequency, but a `GpuTick` is emitted only when a previously unseen device update is detected and can be placed monotonically into Tycho's multi-domain buffer. This separation ensures that increased polling density improves detection latency without inflating the event stream or distorting temporal structure.

Device-level and process-level telemetry are integrated asymmetrically. Device snapshots define the temporal anchor of each `GpuTick`, while process-level records are attached retrospectively to confirmed device updates to accommodate backend-imposed wall-clock windows. This strategy preserves the architectural timing guarantees while enabling multi-tenant attribution under heterogeneous backend constraints.

5.4.4.2 Phase-Aware Sampling Realisation

The phase-aware timing model defined in § 4.5.4 is realised through a conservative observation pipeline that separates sampling attempts from update confirmation. Polling is driven by predicted publish moments, but observations are accepted only when they provide evidence of a previously unseen hardware update. This prevents both aliasing and redundant emission under irregular driver cadence.

Freshness detection prioritises the strongest available backend signal. When reliable cumulative energy counters are present, monotonic advancement of these counters serves as the authoritative indicator of a new publish. On devices lacking such counters, freshness is inferred from instantaneous power changes exceeding a noise-tolerant threshold. In both cases, snapshots that do not satisfy freshness criteria are discarded without affecting estimator state or downstream timelines.

Duplicate suppression is enforced by conditioning all state updates on confirmed freshness. Period and phase estimators are advanced only when a new publish is detected, ensuring that redundant polls neither bias cadence inference nor generate spurious alignment corrections. This guarantees that increased polling density reduces detection latency without inflating the logical event stream.

5.4.4.3 Event Construction and Emission

When a fresh device update is confirmed, the collector constructs a `GpuTick` that represents the accelerator state at a single monotonic timestamp. The device snapshot defines the temporal anchor of the event. If process-level telemetry is available, the corresponding utilisation records, aggregated over a backend-defined wall-clock window, are attached retrospectively to the same tick.

Tick emission is strictly conditional on update confirmation. No `GpuTick` is produced for redundant or ambiguous observations, and no tick is emitted retroactively. Each emitted tick is inserted into Tycho's multi-domain buffer in monotonic order, preserving causal alignment with RAPL, Redfish, and eBPF data without interpolation or reordering.

This construction enforces a one-to-one correspondence between hardware publishes and GPU events. As a result, the GPU timeline reflects device behaviour rather than sampling artefacts and provides a temporally consistent input to subsequent attribution stages.

5.4.4.4 Process Telemetry Integration

Process-level GPU telemetry is exposed by the backend only as utilisation aggregated over an explicit wall-clock interval. This constraint is external to Tycho's timing model and cannot be eliminated at the architectural level. The implementation therefore treats process telemetry as a retrospective signal that must be aligned to, but not conflated with, the device-level event timeline.

To preserve temporal consistency, process queries are issued in conjunction with device polling, but their results are attached only to confirmed device updates. Each process record is associated with the monotonic timestamp of the corresponding device snapshot, establishing a clear temporal anchor without implying instantaneous semantics. Wall-clock durations are tracked independently per device or MIG instance to ensure that backend windows advance correctly regardless of monotonic tick spacing.

Failure handling is deliberately non-blocking. If a process query fails or returns incomplete data, the collector advances the wall-clock origin to avoid repeated zero-length windows, while device-level sampling proceeds unaffected. This ensures that transient backend failures degrade attribution fidelity locally without destabilising cadence inference or event emission.

5.4.4.5 Robustness and Failure Modes

GPU telemetry exhibits substantial variability across hardware generations, driver versions, and backend capabilities. The implementation is therefore designed to

preserve architectural guarantees under incomplete or degraded signals rather than to assume uniform availability.

Missing cumulative energy counters are handled through per-device capability tracking. When authoritative counters are unavailable or non-monotonic, freshness detection falls back to power-based inference with conservative thresholds, preventing noise-induced duplicate events at the cost of increased uncertainty. Backend-specific differences between NVML and DCGM are treated as input variability, not as control flow, ensuring that sampling and emission semantics remain consistent.

Publish cadence jitter is absorbed by the phase-aware inference mechanism. Because estimators are updated only on confirmed publishes, short-term timing irregularities do not propagate into spurious alignment corrections or event duplication. At worst, detection latency increases temporarily, while the one-tick-per-publish invariant remains intact.

MIG instances are handled uniformly as independent telemetry sources during collection. No additional analytical assumptions are introduced at this stage, and MIG metadata is propagated without special treatment. This conservative stance avoids overstating attribution guarantees in configurations where downstream analysis does not explicitly model MIG topologies.

5.4.4.6 Implementation Consequences

The GPU collector implementation enforces the architectural timing guarantees in the presence of implicit publish cadences, heterogeneous backend capabilities, and partial observability. In practice, this ensures that the GPU event stream is free of redundant samples, causally ordered with respect to all other measurement domains, and aligned to genuine hardware updates rather than to polling artefacts. The one-to-one correspondence between confirmed device publishes and emitted `GpuTick` events is preserved even under jitter, missing counters, or transient backend failures.

At the same time, the implementation inherits unavoidable limitations from the telemetry ecosystem. Publish cadence inference is necessarily approximate, and process-level utilisation remains aggregated over backend-defined wall-clock windows. These constraints bound the temporal precision of attribution but do not violate the correctness or ordering guarantees of the GPU timeline.

By producing a temporally consistent, event-driven GPU measurement stream, the collector enables downstream analysis stages to correlate accelerator activity with CPU, memory, and platform power without resampling or heuristic alignment. This integration is a prerequisite for accurate cross-domain attribution and allows later stages to reason about GPU energy consumption under the same invariants that govern all other Tycho subsystems.

5.4.4.7 Collected Metrics

The GPU collector reports both device-level and process-level telemetry for each emitted `GpuTick`. Device metrics capture the instantaneous operational state of the accelerator at the time of a confirmed publish, while process metrics describe aggregated utilisation over the corresponding backend window. Tables 5.4 and 5.5 summarise the metrics collected at each level.

Metric	Unit	Description
Utilisation metrics		
`SMUtilPct`	%	Streaming multiprocessor (SM) utilisation.
`MemUtilPct`	%	Memory controller utilisation.
`EncUtilPct`	%	Hardware video encoder utilisation.
`DecUtilPct`	%	Hardware video decoder utilisation.
Energy and thermal metrics		
`PowerMilliW`	mW	Instantaneous power via NVML/DCGM (1s average).
`InstantPowerMilliW`	mW	High-frequency instantaneous power from NVIDIA field APIs.
`CumEnergyMilliJ`	mJ	Cumulative energy counter (preferred freshness signal).
`TempC`	°C	GPU temperature.
Memory and frequency metrics		
`MemUsedBytes`	bytes	Allocated framebuffer memory.
`MemTotalBytes`	bytes	Total framebuffer memory.
`SMClockMHz`	MHz	SM clock frequency.
`MemClockMHz`	MHz	Memory clock frequency.
Topology and metadata		
`DeviceIndex`	–	Numeric device identifier.
`UUID`	–	Stable device UUID.
`PCIBusID`	–	PCI bus identifier.
`IsMIG`	–	Indicates a MIG instance.
`MIGParentID`	–	Parent device index for MIG instances.
`Backend`	–	Backend type (`NVML` or `DCGM`).

TABLE 5.4: Device- and MIG-level metrics collected by the GPU subsystem.

Metric	Unit	Description
Per-process utilisation metrics		
`Pid`	–	Process identifier.
`ComputeUtil`	%	Per-process SM utilisation aggregated over the query window.
`MemUtil`	%	Per-process memory controller utilisation.
`EncUtil`	%	Per-process encoder utilisation.
`DecUtil`	%	Per-process decoder utilisation.
Device and timing metadata		
`GpuIndex`	–	Device or MIG instance to which the sample belongs.
`GpuUUID`	–	Corresponding device UUID.
`TimeStampUS`	µs	Backend timestamp associated with the utilisation record.
MIG metadata (when applicable)		
`GpuInstanceID`	–	MIG GPU instance identifier.
`ComputeInstanceID`	–	MIG compute-instance identifier.

TABLE 5.5: Process-level metrics collected over a backend-defined time window.

5.5 Metadata and Identity Infrastructure

5.5.1 Architectural Context

This section realises the metadata subsystem defined in § 4.6 as a refresh-driven, bounded cache that supplies joinable workload identity to the analysis engine. Metadata does not form a time series and is not evaluated over analysis windows. Its function is to provide sufficiently fresh identity state at analysis boundaries while remaining robust under partial observability, workload churn, and asynchronous sources.

5.5.2 Controller-Orchestrated Refresh and Lifetime Enforcement

All metadata mutation is centralized in a metadata controller, which constitutes the sole authority over state updates and lifecycle management. Collectors never modify analysis-visible state directly and never delete entries. They submit observations to the controller, which serializes updates and enforces freshness and retention rules. This separation is required to ensure that identity state is maximally fresh at analysis boundaries while remaining bounded and deterministic under missing or partial updates.

At the start of every analysis cycle, the analysis engine triggers exactly one metadata refresh. This refresh dominates all other scheduling and ensures that identity state reflects the most recent observable system structure at the moment the analysis window is evaluated. Additional refreshes within the same cycle are unnecessary, as the window is closed at the cycle boundary and later identity changes cannot affect its evaluation.

To bound metadata age when analysis cycles are long or sparse, the controller may execute collectors periodically. Each collector has an independent freshness target, with defaults of 1 s for the proc collector and 3 s for the kubelet collector. Background execution is suppressed when an analysis-triggered refresh is imminent, specifically when it lies within 250 ms, ensuring that the cycle-start refresh dominates and that redundant collection near analysis boundaries is avoided.

Metadata collection is explicitly best-effort. Collectors do not define a notion of success and may submit partial updates. The controller accepts all updates without requiring a complete snapshot, and cache convergence is achieved through repeated refreshes. Missing observations are handled exclusively through horizon-based expiry rather than through collection-time interpretation, preserving a strict separation between identity acquisition and attribution logic.

5.5.3 Metadata Store, Keys, and Temporal Alignment

Metadata is stored in an in-memory cache partitioned by entity type. Entries represent the most recent known identity state and are overwritten in place on update; historical versions are not retained. Pod entries are keyed by pod UID, container entries by normalized runtime container ID, and process entries by PID with the process start token (`StartJiffies`) retained for disambiguation.

PID reuse is handled by treating the pair (PID, `StartJiffies`) as the effective process identity. When a PID is reused, a new entry is created rather than overwriting

the prior instance, preventing stale process metadata from being joined with unrelated activity during overlapping analysis windows.

All metadata updates within a refresh are timestamped once using Tycho's global monotonic timebase. The same timestamp is applied uniformly to all entries updated in that refresh, ensuring consistent temporal alignment with metric observations without introducing enumeration-induced skew. An auxiliary wall-clock timestamp is recorded for diagnostics but is not used in attribution logic.

Garbage collection is executed periodically by the controller and independently of refresh scheduling. Entries whose last-seen timestamp falls outside the same retention horizon used for metric buffers are removed. This alignment guarantees that any retained metric observation remains joinable with identity metadata while providing deterministic memory bounds and natural cache drainage under missing updates.

5.5.4 Proc Collector

The proc collector provides the operating-system view required to associate process-level activity with container identity. It enumerates processes via the proc filesystem and records a deliberately minimal attribute set consisting of process identifiers, command name, and cgroup membership. This minimalism avoids unstable or expensive enrichment at collection time while retaining all information required for later joins.

Process-to-container association is derived from cgroup membership and normalized into a runtime container identifier. Both cgroup version 1 and version 2 layouts are supported, and normalization targets containerd and CRI-O runtimes. Processes that cannot be mapped to a Kubernetes container are labeled with a sentinel container identifier stored directly in the process entry, avoiding synthetic container objects and keeping system activity explicit at analysis time.

Process enumeration is inherently racy. Processes may disappear during traversal and individual reads may fail due to lifecycle races. Such failures are treated as normal; only successfully read entries are updated, and stale state is removed by horizon-based expiry.

5.5.5 Kubelet Collector

The kubelet collector supplies the authoritative Kubernetes node-local view required for correct pod and container attribution. It periodically retrieves the kubelet `/pods` endpoint and persists only identity information that cannot be reliably reconstructed after termination.

Container identifiers are normalized at collection time, and only the normalized form is stored. Init containers and ephemeral containers are represented uniformly as container entries with lifecycle-dependent status categories. Termination state and exit codes are recorded when available, allowing analysis to avoid attributing energy to completed workloads without relying on event histories.

The kubelet view is the sole stable source of resource specifications once workloads terminate. Tycho therefore stores CPU and memory requests and limits for both containers and aggregated pods. If the kubelet is temporarily unreachable, no updates are applied for that refresh. There is no explicit freshness gating in analysis; degradation manifests through missing or stale joins bounded by the retention horizon.

5.5.6 Metadata Contract and Join Surface

The metadata subsystem exposes a fixed set of identity fields that define the complete join surface available to the analysis engine. Tables 5.6, 5.7, and 5.8 summarize the fields collected by the proc and kubelet collectors. This inventory constitutes an implementation contract: attribution feasibility and correctness depend directly on the presence and semantics of these fields, and no additional identity information is assumed downstream.

Field	Source	Description
Process identity		
PID	`/proc`	Numeric process identifier; unique at any moment but reused over time.
StartJiffies	`/proc/<pid>/stat`	Kernel start time of the process in clock ticks (jiffies), used to detect PID reuse.
Container and system classification		
Container ID	Kepler cgroup resolver	Normalized container identifier for pod processes; `system_processes` for host and kernel processes.
Command	`/proc/<pid>/comm`	Short command name for debugging and manual inspection.
Timestamps		
LastSeenMono	Monotonic timebase	Timestamp aligned with metric collectors.
LastSeenWall	Controller timestamp	Wall-clock timestamp for GC.

TABLE 5.6: Process metadata collected by the process collector

Field	Source	Description
Pod identity		
PodUID	Kubelet PodList	Stable pod identifier for correlation and container grouping.
PodName, Namespace	Kubelet PodList	Human-readable pod identity and namespace.
Lifecycle and scheduling context		
Phase	PodStatus	Coarse pod state (Pending, Running, Succeeded, Failed).
QoSClass	PodStatus	Kubernetes QoS classification (Guaranteed, Burstable, BestEffort).
OwnerKind / OwnerName	Pod metadata	Controller reference (e.g. ReplicaSet, DaemonSet).
Resource specifications		
Requests (CPU, Memory)	`pod.spec.containers`	Aggregate pod-level requests following Kubernetes scheduling semantics.
Limits (CPU, Memory)	`pod.spec.containers`	Aggregate pod-level limits following Kubernetes scheduling semantics.
Timestamps		
LastSeenMono	Monotonic timebase	Timestamp aligned with metric collectors.
LastSeenWall	Controller timestamp	Wall-clock timestamp for GC.

TABLE 5.7: Pod metadata collected by the kubelet collector

Field	Source	Description
Container identity		
ContainerID	PodStatus	Normalized container identifier.
ContainerName	PodStatus	Declared container name within pod.
Lifecycle state		
State	ContainerStatus	Fine-grained state (Running, Waiting, Terminated).
ExitCode	ContainerStatus	Termination exit code when available.
Resource specifications		
Requests (CPU, Memory)	`pod.spec.containers`	Container-level resource requests; preserved for terminated containers.
Limits (CPU, Memory)	`pod.spec.containers`	Container-level resource limits.
Timestamps		
LastSeenMono	Monotonic timebase	Timestamp aligned with metric collectors.
LastSeenWall	Controller timestamp	Wall-clock timestamp for GC.

TABLE 5.8: Container metadata collected by the kubelet collector

5.5.7 Design Consequences and Exclusions

By centralizing lifecycle enforcement and treating sources as independently valid, the subsystem provides maximally fresh identity at analysis boundaries while remaining robust to partial observability and transient failures. Correctness relies on bounded freshness and explicit joins rather than on point-in-time snapshots or event histories.

5.6 Calibration

This section describes how the calibration architecture introduced in § 4.7 is realized in Tycho's implementation. Calibration is implemented as a bounded, startup-time procedure whose sole role is to reduce temporal uncertainty in hardware-controlled metric publication before steady-state collection begins. It exists to improve the correctness of downstream timing and attribution under partial observability, without introducing runtime adaptation or feedback.

Calibration is optional and can be disabled via configuration. When enabled, it is executed at most once per Tycho process lifetime and is not re-entered or repeated. All calibration results are node-local and apply only to the process instance that performed them.

5.6.1 Architectural Context

The calibration architecture distinguishes two orthogonal concerns: polling-frequency calibration, which bounds the minimum safe polling period for hardware-controlled metric sources, and delay calibration, which bounds the reaction latency between workload transitions and observable metric changes. Only polling-frequency calibration is integrated into Tycho's startup path. Delay calibration is treated as an external preparatory step and is consumed purely as static configuration.

Calibration parameters are consumed by the timing and analysis subsystems. They do not influence runtime attribution logic directly and do not observe live workloads.

5.6.2 Startup Strategy and Collector Gating

Polling-frequency calibration is executed during Tycho startup, prior to enabling the affected collectors. Collectors whose polling cadence depends on calibrated bounds are held inactive until calibration completes or is bypassed. As a consequence, no metrics from these collectors are emitted before a polling interval has been selected.

Calibration does not block system startup indefinitely. If insufficient observations are obtained or calibration fails for any reason, Tycho falls back to user-configured default polling intervals. This fallback is silent at the semantic level and does not alter runtime behavior, ensuring that metric availability is not contingent on successful calibration. Empirically derived bounds are preferred when available, but correctness does not depend on their presence.

When multiple devices contribute to a single collector on a node, calibration results are aggregated conservatively. The most restrictive bound across all observed devices is selected and applied uniformly, ensuring that no device-level publication is undersampled due to intra-node variability.

5.6.3 Polling-Frequency Calibration Mechanism

Polling-frequency calibration is realized as a short-lived hyperpolling phase. During this phase, Tycho queries the relevant hardware interface at a conservatively high rate and passively observes the arrival of distinct metric updates. From these observations, it derives a conservative bound on the minimum safe polling period that avoids undersampling under nominal conditions.

For GPU power metrics, calibration is performed independently for each device using NVML. Per-device observations are aggregated at node level, and the most restrictive bound is selected. The resulting polling period is then applied uniformly to all GPU collectors on that node, ensuring consistent temporal coverage across heterogeneous devices.

For Redfish power metrics, calibration operates at the level of the BMC. Observed updates across all exposed chassis contribute to the inferred cadence, allowing calibration to remain valid in multi-chassis configurations. When Redfish *heartbeat* is enabled, the calibrated polling period is additionally constrained by a hard cap derived from the heartbeat interval. Calibration therefore establishes a lower bound on safe polling, while heartbeat logic enforces upper limits on staleness during steady-state operation. The two mechanisms are strictly layered and do not overlap in responsibility.

No polling-frequency calibration is performed for RAPL or eBPF. RAPL counters update quasi-continuously relative to Tycho's temporal resolution, making undersampling architecturally irrelevant. eBPF metrics are event-driven and decoupled from device-side publication cadence, rendering polling-frequency discovery unnecessary.

5.6.4 Delay Calibration Integration

Delay calibration is not performed within Tycho. Estimating the latency between workload transitions and observable metric reactions requires controlled, high-intensity workload generation that is incompatible with Tycho's non-intrusive monitoring constraints.

Instead, delay calibration is carried out offline using external tooling that executes directly on the target system. These measurements derive conservative, device-specific delay bounds for GPU power metrics. The resulting bounds are supplied to Tycho exclusively via configuration. At runtime, Tycho treats these bounds as static parameters and applies them during analysis to prevent premature attribution and to align metric data with workload phases.

When multiple GPU devices are present, Tycho selects the smallest configured delay bound across devices and applies it uniformly. This choice favors temporal responsiveness while remaining consistent with the best-effort nature of delay estimation.

No delay calibration is applied to Redfish. Irregular publication intervals, variable network latency, and opaque BMC-internal behavior preclude stable delay estimation. Redfish metrics are therefore treated as coarse signals whose temporal coherence is enforced through freshness tracking and scheduling constraints rather than delay correction.

5.6.5 Implementation Consequences

Calibration improves attribution correctness by reducing avoidable temporal uncertainty in hardware-controlled metric sources. It provides best-effort bounds that constrain polling and alignment decisions without introducing runtime adaptation or control coupling. By resolving these uncertainties ahead of steady-state operation, Tycho preserves deterministic execution, bounded timing behavior, and strict separation between observation and analysis.

5.7 Analysis and Attribution Infrastructure

5.7.1 Analysis Engine Responsibilities and Cycle Lifecycle

This section describes how the orchestration model defined in § 4.8.1 is enforced at runtime. The analysis engine is a minimal orchestration authority whose sole responsibility is to execute window-scoped attribution deterministically and without hidden coupling. It constructs analysis cycles, selects attribution windows, enforces a fixed execution order, and establishes a materialization boundary for derived results. The engine does not participate in metric semantics, dependency inference, modeling decisions, result interpretation, or exporter interaction.

Each invocation of the engine corresponds to exactly one analysis cycle. Cycles are triggered externally by the timing engine; the analysis engine does not self-schedule, maintain timers, or perform background work. Invocation cadence and jitter are therefore treated as external concerns and do not affect correctness, provided that invocations are serialized and the monotonic timebase is strictly increasing.

At cycle entry, the engine atomically constructs a fresh execution context that fully determines the behavior of the cycle. This context includes the selected attribution window, admissibility constraints for reading observations, read-only access to upstream buffers, access to node-local metadata, access to shared cross-window state, and a cycle-local materialization boundary. No additional execution context is injected after construction, and no global mutable analysis state is consulted during execution.

Once constructed, the cycle is executed exactly once. The engine invokes all metrics sequentially according to a fixed execution plan. There is no re-entry into a partially executed cycle, no mid-cycle rescheduling, and no orchestration-level retry or backtracking. After execution completes, all cycle-local structures are discarded, including the materialization store, and no derived quantities persist implicitly into subsequent cycles.

Failures are isolated to the cycle in which they occur. If an individual metric fails to produce a result, the failure is logged and execution continues for remaining metrics. Such failures affect only the completeness of results for the current window and do

not abort the cycle. Each subsequent invocation constructs a new cycle with a fresh execution context, independent of prior errors or partial results.

By enforcing atomic cycle construction, single-pass execution, and explicit teardown, the engine guarantees that each cycle yields at most one coherent set of window-scoped results. There is no opportunity for partial publication, retroactive correction, or cross-cycle interference. This execution discipline provides the structural foundation on which staging, materialization, and cross-window modeling are implemented while preserving determinism, auditability, and non-retrospective semantics.

5.7.2 Attribution Window Selection and Temporal Safety

This section describes how the attribution window defined in § 4.8.1.3 is selected and enforced at runtime. Window selection is performed exactly once at cycle entry and is derived exclusively from the global monotonic timebase, which serves as the sole temporal authority. Wall-clock time, collector timestamps, and exporter behavior are not consulted, ensuring a strictly ordered and unambiguous temporal reference.

The window end t_k is selected with a fixed intentional lag relative to real-time execution. This lag corresponds to the maximum admissible metric delay plus a safety margin obtained from configuration and optional calibration. By construction, each window is therefore placed in a region of the past where all participating metrics are guaranteed to have observed the samples required to interpret their contributions under their declared delay semantics. The lag is constant across cycles, yielding attribution windows with fixed duration and uniform semantics in steady state.

During startup, when insufficient observation history exists to populate the intended window fully, the window start may be clamped to the beginning of the monotonic timeline. This behavior is explicit, deterministic, and confined to the initial phase of execution. No other deviations from the steady-state window definition are permitted.

Window selection is deliberately decoupled from collector sampling behavior. Collectors may operate at different frequencies, with irregular timing or transient gaps, without influencing window boundaries. The engine does not impose an alignment grid or attempt to synchronize with sampling schedules. Instead, windows are defined purely in terms of monotonic ticks, and metrics interpret the contents of upstream buffers over the selected interval using interval semantics and metric-local delay correction.

Once selected, the attribution window is immutable for the duration of the cycle. All metrics observe the same base window, and any effective windows derived through delay correction are computed deterministically from this reference. The engine does not revise window boundaries mid-cycle and does not reopen or reinterpret windows after execution.

By combining a monotonic timebase, a fixed safety lag, and immutable window selection, the implementation enforces temporal admissibility by construction. Attribution correctness is insulated from collector jitter and delayed observations, speculative window closure is avoided, causality is preserved, and each cycle yields a single, final temporal interpretation of the available evidence.

5.7.3 Staged Pipeline Execution and Dependency Discipline

This section explains how the dependency discipline defined in § 4.8.1.4 is enforced at runtime. Dependency correctness is not inferred dynamically and is not validated during execution. Instead, it is achieved by construction through a fixed execution order that reflects the semantic structure of the analysis pipeline.

For each analysis cycle, the engine constructs a static execution plan consisting of an ordered list of metric invocations. This plan is invariant across cycles for a given build and configuration. Metrics are registered into the analysis registry in an order that encodes their semantic dependencies, and the engine preserves this order exactly during execution. As a result, metrics executed later in the plan may depend on outputs produced earlier in the same cycle, while reverse dependencies are structurally impossible.

Stages are not represented as explicit runtime entities. Instead, stage boundaries are implicit and correspond to contiguous segments of the execution plan. Each segment groups metrics that operate at the same semantic level, such as aggregation, decomposition, or attribution. Downstream segments assume that all metrics in earlier segments have either materialized their window-scoped outputs or abstained from doing so due to missing or inadmissible inputs.

The implementation deliberately avoids constructing dependency graphs, performing topological sorting, or validating dependency satisfaction at runtime. There are no per-metric dependency declarations and no control flow conditioned on the availability of upstream results. If a metric observes an undefined input, it degrades according to its semantics rather than triggering reordering, backtracking, or failure.

This design treats dependency correctness as an architectural obligation rather than an algorithmic problem. The runtime enforces execution order faithfully but does not attempt to infer semantic validity. By resolving dependencies explicitly at composition time, the implementation preserves determinism, avoids hidden coupling between metrics, and ensures that degradation under partial observability propagates monotonically through the pipeline without compromising internal consistency.

5.7.4 Metric Materialization and Intra-Cycle Visibility

This section describes how the materialization model is enforced at runtime. At the beginning of each analysis cycle, the engine establishes a cycle-local materialization boundary that serves as the sole authoritative representation of all derived results for the current attribution window. All metric emissions during the cycle are routed through this boundary, ensuring that materialization is strictly window-scoped and isolated from other cycles.

Derived results are recorded as immutable, window-scoped facts. Within a cycle, each metric is expected to materialize at most one final value per metric identity and label set. Multiple emissions for the same identity deterministically overwrite earlier ones, but such behavior is not relied upon for correctness. The effective semantic contract is therefore exactly-once materialization per window.

All materialized results are conceptually visible to metrics executed later in the same

cycle. There is no notion of stage-local visibility or scoped access control. Dependency correctness relies entirely on execution order rather than on restricting access to intermediate results. Metrics that consume undefined inputs must degrade according to their semantics rather than delaying execution or attempting recovery.

Materialized results never persist implicitly across cycles. When a cycle completes, the materialization boundary is discarded in its entirety, and all derived quantities cease to exist from the perspective of subsequent cycles. Any dependence on prior windows must be mediated through explicit cross-window state owned and managed by the consuming metric.

By enforcing a per-cycle materialization boundary with global intra-cycle visibility and strict teardown semantics, the implementation guarantees that each cycle yields a single, final, and internally consistent set of derived quantities for its attribution window. This prevents hidden cross-window coupling, eliminates incremental refinement of results, and allows later attribution stages to operate on materialized facts rather than raw observations while preserving determinism and auditability.

5.7.5 Cross-Window State and Explicit Memory

This section describes how limited cross-window memory is supported without violating the window-scoped and non-retrospective execution model defined in § 4.8. While derived results never persist across cycles, certain attribution models require controlled statefulness in order to converge or adapt over time. The implementation accommodates such models through an explicit and narrowly scoped state mechanism.

Cross-window state is provided through a shared state store that persists for the lifetime of the analysis process and is made available to each cycle at construction time. The state store is passive and provides no execution logic or semantic interpretation. The analysis engine does not read from, write to, or reason about its contents; its sole responsibility is to supply access to metrics during cycle execution.

All cross-window state is strictly metric-owned. Metrics that require memory across windows must explicitly retrieve, initialize, and update their own state entries. Metrics that do not access the state store remain purely window-scoped and stateless by construction. State entries are keyed by metric identity and labels, making ownership and scope explicit and preventing implicit sharing between unrelated metrics.

Use of cross-window memory is an explicit opt-in. Any temporal coupling introduced by stateful behavior is therefore visible at the point of use and auditable in isolation. State is updated only after a cycle completes and influences only subsequent cycles. Previously materialized results are never revised, and no stateful mechanism can retroactively alter the interpretation of earlier windows.

By confining cross-window memory to explicit, metric-owned state and maintaining strict cycle boundaries, the implementation supports stateful attribution models where required, while preserving determinism, isolation, and the architectural guarantee that each cycle yields a single, final interpretation of its attribution window.

5.7.6 Output Commit and Sink Boundary

This section describes how analysis results are committed and published without allowing downstream mechanisms to influence attribution semantics. At cycle construction time, the engine establishes a collecting sink that serves as the sole emission boundary for all metrics executed during the cycle. All derived results emitted by metrics are routed through this boundary and are treated as belonging to the same attribution window.

Result commitment occurs at the moment of materialization. Once a derived quantity is recorded within the cycle-local materialization boundary, it is final for the current window. Publication to downstream sinks is strictly observational and occurs after materialization without providing feedback into the analysis process. The engine never reads from sinks, waits for acknowledgements, or conditions execution on publication success.

Sinks are therefore non-authoritative by design. Exporter behavior may delay, drop, aggregate, or reorder published results, but such behavior does not alter the meaning or validity of the computed window-scoped quantities. Attribution correctness is defined entirely within the analysis layer and is independent of downstream reliability or performance characteristics.

All results produced during a cycle are committed as a logical batch. There is no partial publication of intermediate results and no distinction between provisional and final outputs. Once cycle execution completes, the set of materialized results represents the maximal, internally consistent interpretation achievable for the attribution window.

By strictly separating analysis from publication, the implementation preserves reproducibility and robustness. Failures at the export layer degrade only the visibility of results, not their correctness, and cannot introduce hidden coupling or timing dependencies into the attribution process. This boundary ensures that attribution semantics remain stable, auditable, and invariant under changes to exporter implementation or behavior.

5.7.7 Implementation Consequences and Guarantees

The implementation described in this chapter enforces the architectural contract of the analysis layer directly through execution structure rather than dynamic control logic. Each analysis cycle yields a single, deterministic, window-scoped interpretation of the available evidence, constructed under fixed temporal assumptions and executed in a strictly ordered, non-retrospective manner. All derived results are materialized explicitly, remain immutable for the duration of the cycle, and are isolated from both prior and subsequent cycles.

Temporal correctness is ensured by construction. Attribution windows are selected once per cycle from a global monotonic timebase using a fixed safety lag, are immutable during execution, and are interpreted without reliance on collector sampling grids or exporter timing. As a result, causality is preserved and attribution semantics are insulated from collection jitter, delayed observations, and downstream publication behavior.

Dependency correctness is enforced structurally. Metric execution order is fixed and reproducible, stages are implicit in the execution plan, and no runtime dependency inference or reordering occurs. Under partial observability, results degrade monotonically through omission or explicitly defined fallback semantics, and previously materialized interpretations are never revised.

Stateful attribution models are supported exclusively through explicit, metric-owned cross-window state. No derived quantities persist implicitly across cycles, and any temporal coupling introduced by memory is localized, visible, and auditable. This preserves isolation between cycles while allowing convergence or adaptation where required.

Equally important are the properties the implementation does not guarantee. The analysis layer does not ensure completeness of results for every window, does not backfill or reinterpret prior windows, does not validate semantic correctness of pipeline composition at runtime, and does not provide reliability guarantees for result publication. These non-guarantees are deliberate and reflect the prioritization of correctness, determinism, and transparency over forced coverage or convenience.

Together, these consequences establish a stable execution contract for attribution. They enable later analysis stages to operate on materialized, window-scoped quantities with well-defined temporal and dependency semantics, while ensuring that increasing analytical sophistication does not compromise determinism, auditability, or adherence to the architectural model.

5.7.8 Stage 1: Component Metric Construction

5.7.8.1 Component-Level eBPF Utilization Metrics (Totals)

Architectural Context. This subsubsection realizes the utilization metrics defined in § 4.8.2.1. It implements the window-aligned construction of node-level CPU time-share ratios and cumulative kernel and hardware counters from discrete eBPF observations, without redefining architectural semantics or introducing additional modeling assumptions.

Implementation Strategy. Discrete eBPF observations are consumed as per-tick deltas. For each analysis window, deltas are integrated over an effective window interval to approximate the continuous quantities defined architecturally. CPU execution time is normalized against node capacity to obtain ratios, while all other eBPF-derived signals are aggregated across processes and accumulated into persistent node-level counters.

Core Mechanisms. Per-tick deltas for CPU idle, hardware interrupt, and software interrupt execution are integrated independently over the window. Normalization uses the product of window duration and logical CPU count to obtain dimensionless time-share ratios. Active CPU utilization is derived as the complement of the integrated non-active components, enforcing exact conservation of schedulable CPU capacity. For event-based signals, per-process deltas are summed at each tick, integrated over the window, and added to cumulative counters that advance monotonically across analysis cycles.

Robustness and Edge Cases. Window boundaries may intersect eBPF tick intervals. Integration therefore accounts for partial overlap between ticks and windows to preserve conservation and avoid bias at window edges. Internal accumulation admits fractional contributions to accommodate overlap, while exported counters are materialized as integer-valued quantities. If insufficient eBPF samples are available to conservatively integrate a window, no metrics are emitted for that window.

Implementation Consequences. The implementation enforces the architectural guarantees of strict monotonicity for counters and exact partitioning of CPU capacity across utilization classes. All cumulative metrics are scoped to the lifetime of the Tycho process and may reset on restart. These semantics ensure stable interpretation under sampling variability and allow downstream stages to rely on consistent utilization signals.

Exported Metrics. The metrics exported by this implementation are summarized in the table 5.9. Only fully materialized utilization metrics intended for external consumption are listed.

Metric	Type	Unit	Labels
CPU time-share ratios			
`bpf_cpu_idle_ratio`	Gauge	ratio	`source`
`bpf_cpu_irq_ratio`	Gauge	ratio	`source`
`bpf_cpu_softirq_ratio`	Gauge	ratio	`source`
`bpf_cpu_active_ratio`	Gauge	ratio	`source`
Aggregated cumulative counters			
`bpf_cpu_instructions`	Counter	count	`source`
`bpf_cpu_cycles`	Counter	count	`source`
`bpf_cache_misses`	Counter	count	`source`
`bpf_page_cache_hits`	Counter	count	`source`
`bpf_irq_net_tx`	Counter	count	`source`
`bpf_irq_net_rx`	Counter	count	`source`
`bpf_irq_block`	Counter	count	`source`

Label domain: `source` = `bpf`.

TABLE 5.9: Exported utilization metrics derived from `eBPF` observations.

5.7.8.2 RAPL Component Metrics (Totals)

Architectural Context. This section realises the cumulative RAPL energy model defined in the corresponding architecture subsubsection, producing zero-based, monotonic energy counters per RAPL domain and an auxiliary, window-averaged power signal. Only total energy and total power are constructed here; no decomposition or attribution is performed at this stage.

Implementation Strategy. The implementation consumes time-ordered RAPL counter samples within the effective analysis window and aggregates them across sockets for each domain. Energy counters are constructed directly from native cumulative readings, while power is derived secondarily from in-window energy differences. All logic is structured to ensure that exported energy counters remain monotonic, zero-based, and independent of window boundaries.

Core Mechanisms. For each domain, the implementation identifies the first and last available cumulative RAPL counters within the effective window. The last counter represents the native cumulative energy at window end. To eliminate hardware-defined initial offsets, the first observed native value per domain is stored once and subtracted from all subsequent exports, yielding a zero-based cumulative energy counter.

Window-local energy increments are computed by differencing the first and last counters within the window. Because raw RAPL inputs are already corrected for wraparound upstream, no additional wraparound handling is required at this stage. Average power is then derived by dividing the in-window energy increment by the window duration. This power signal is emitted alongside the energy counter but is not used for any downstream computation.

Robustness and Edge Cases. If fewer than two valid RAPL samples are available within a window, no metrics are emitted, avoiding partial or misleading updates. Non-positive or ill-defined window durations suppress emission entirely. All aggregation is performed per domain and summed across sockets conservatively, ensuring that missing socket data cannot inflate reported energy. Since cumulative energy is always derived from native counters and offset subtraction is monotonic, exported energy values never decrease.

Implementation Consequences. The implementation guarantees that exported RAPL energy metrics are stable cumulative counters suitable as authoritative inputs for later stages. Auxiliary power metrics are provided solely for inspection and convenience and are explicitly excluded from attribution logic. Optional quality or diagnostic metadata may be emitted for debugging purposes, but it does not affect metric semantics.

Exported Metrics. The metrics exported by this implementation are summarized in the table 5.10. Only cumulative energy counters intended as authoritative inputs and their auxiliary, user-facing power counterparts are listed.

Metric	Type	Unit	Labels
RAPL component totals (per domain)			
`rapl_energy_mj`	Counter	mJ	`domain, kind, source`
`rapl_power_mw`	Gauge	mW	`domain, kind, source`

Label domain: `domain` $\in$ {pkg, core, uncore, dram}, `kind` = total, `source` = `rapl`.

TABLE 5.10: Exported RAPL component energy and power metrics.

5.7.8.3 GPU Component Metrics (Totals)

Architectural Context. This section realizes the architectural definition of GPU power and energy reconstruction as a history-aware, constraint-based process. Unlike collectors that derive window-local quantities directly from raw samples, the

GPU metric maintains a corrected power signal over a retained corrected-time horizon and derives windowed quantities as projections of that maintained signal. Historical state is therefore treated as a correctness mechanism rather than an optimization, enabling reconciliation of delayed, averaged, and cumulative GPU observations into a single authoritative timeline. All execution-time mechanisms described below exist to uphold this contract under discretization, partial observability, and bounded computation.

Implementation Strategy. GPU reconstruction is implemented by maintaining a per-device corrected power series on a uniform corrected-time grid that persists across analysis cycles. Rather than recomputing power independently per window, the implementation incrementally extends and updates this series using newly available observations while preserving previously reconstructed history. To bound computation, reconstruction is confined to a moving tail region near the current analysis window, while constraints are derived from observations spanning a longer retained history. Windowed GPU energy and power are obtained by projecting the maintained series onto the analysis window, ensuring temporal coherence across windows and alignment with the accuracy-first architectural objective.

Constraint Realization. Raw GPU telemetry is realized as weighted constraints on the corrected power series. Instantaneous power samples contribute point-wise soft constraints, while one-second averaged power samples are realized as boxcar-mean constraints over the preceding approximately one-second interval on the uniform grid. Cumulative energy counters, when present and validated, contribute dominant consistency constraints derived from positive, monotonic energy deltas mapped to the corresponding grid intervals. Constraint families use fixed internal relative weights reflecting expected reliability, with cumulative energy dominating when enabled. All constraints act jointly on the reconstructed series in corrected time, avoiding sequential correction or signal prioritization.

Reconstruction and Update Semantics. Reconstruction is performed on a moving tail of the corrected power history covering the most recent portion of the retained horizon. Only this tail segment is re-solved each cycle, after which it overwrites the corresponding region of the maintained history on the uniform grid. This moving-horizon approach bounds computational cost while allowing historical observations to influence the present reconstruction. If reconstruction cannot be performed due to insufficient or inconsistent observations, the previously retained history is preserved unchanged, ensuring continuity of the corrected signal across cycles.

Windowed Energy and Power Derivation. Windowed GPU energy is derived by projecting the corrected power history onto the current analysis window. When validated cumulative energy counters are available and a prior baseline exists, the window energy increment is obtained directly from the counter delta to preserve absolute energy consistency. Otherwise, window energy is computed by integrating the corrected power series over the window interval. Windowed GPU power is derived from the corresponding window energy and duration and represents the same underlying quantity as a gauge. Per-window energy values are treated as auxiliary quantities, while the maintained reconstructed history remains authoritative.

Robustness and Edge Cases. The implementation explicitly tolerates missing, delayed, and partially invalid GPU telemetry. Cumulative energy constraints are enabled only when counters are present, non-zero, strictly increasing, and yield valid deltas; otherwise they are disabled automatically. Non-negativity is enforced via post-solve projection to prevent physically implausible power estimates. If projection affects a substantial fraction of bins, a conservative second reconstruction pass is attempted with cumulative energy constraints disabled, and the better-conditioned solution is retained. Numerical conditioning is ensured through a minimal diagonal stabilization term applied purely for solver robustness and without modeling semantics. Transient reconstruction failures preserve the previously retained history, preventing discontinuities in downstream metrics.

Implementation Consequences. By maintaining a corrected GPU power history across analysis cycles, the implementation achieves temporal coherence and energy consistency that window-local estimation cannot provide. Historical context enables delayed, averaged, and cumulative observations to jointly constrain the reconstructed signal, improving stability and effective temporal resolution under realistic telemetry conditions. When cumulative energy counters are available, absolute energy consistency is preserved across windows; otherwise the system degrades gracefully to history-informed integration without violating physical plausibility. While no claim is made to recover ground-truth GPU power, all exported GPU energy and power metrics are guaranteed to derive from a single, internally consistent reconstructed signal that maximally exploits available information.

Exported Metrics. The exported metrics are summarized in Table 5.11.

Metric	Type	Unit	Labels
Total GPU metrics			
`gpu_energy_mj`	Counter	mJ	`gpu_uuid, kind, source`
`gpu_power_mw`	Gauge	mW	`gpu_uuid, kind, source`

Label domain: `kind` = total, `source` = `nvml_corrected`.

TABLE 5.11: Exported GPU metrics derived from the corrected reconstruction.

5.7.8.4 Redfish Component Metrics (Totals)

Architectural Context. This section realises the Redfish-based system power construction defined in § 4.8.2.4. The architecture specifies two complementary system-level series: a raw integration of Redfish telemetry and a corrected reconstruction that treats Redfish as an anchoring signal for a higher-rate proxy-based model. The implementation must therefore support delay-aware observation handling, horizon-spanning state, adaptive alignment, and a controlled transition from raw to corrected emission without violating counter continuity.

Implementation Strategy. The implementation is split into three cooperating responsibilities. First, a raw producer integrates sparse Redfish samples over the effective analysis window and maintains the canonical system counter during warmup. Second, a fusion substrate maintains a fixed-grid horizon cache populated from

component proxy metrics and refreshes Redfish observations under a selected effective delay. Third, a corrected producer fits reconstruction parameters against the cached observations, reconstructs per-bin system power, and integrates the reconstruction over the current window. A readiness flag stored in analysis state governs takeover: raw emission remains authoritative until corrected reconstruction is feasible for the selected chassis.

Core Mechanisms. Raw system metrics are produced by selecting all Redfish samples overlapping the delay-shifted effective window, including at most one predecessor sample to enable zero-order-hold integration. Window energy is obtained by integrating held power values over the window extent, and the exported cumulative counter is advanced monotonically in persistent state.

Corrected reconstruction operates on a fixed fusion grid with configurable bin width and horizon length. A horizon cache stores per-bin proxy features derived from CPU package energy, DRAM energy, GPU energy, and CPU instruction counts. Only newly required bins are populated each cycle, while overlapping cache content is preserved by shifting the horizon forward.

Redfish observations are projected into the corrected time domain by subtracting a candidate delay. An adaptive delay is selected by searching a bounded candidate set and scoring each candidate by the extent to which proxy-derived parts power exceeds Redfish power over a recent horizon segment, subject to an explicit noise margin. Delay updates are rate-limited between cycles, with a higher allowable slew when a step change is detected in the proxy power series. The selected delay is persisted in state and reused as a fallback when the search fails.

Model fitting is performed over the horizon using weighted least squares in scaled space. The reconstruction combines proxy features into per-bin system power using non-negative coefficients for physically interpretable terms and an unconstrained instruction-rate coefficient. When non-negativity constraints become active, a constrained refit is performed to reduce bias from post-fit truncation. Reconstructed bin power is integrated over the overlap with the current analysis window to obtain window energy and average power.

To preserve counter continuity at takeover, the corrected cumulative counter is represented as the sum of a local corrected accumulator and a one-time offset. The offset is seeded from the last available raw exported counter when corrected emission becomes active, ensuring that the canonical counter remains continuous across the transition.

Robustness and Edge Cases. Corrected emission is suppressed until a sufficient number of usable Redfish observations exist within the horizon. Observations whose corrected timestamps would underflow are discarded to prevent degenerate kernel projections and unbounded bin scans. If proxy features are unavailable or the delay-selection search fails, the implementation conservatively retains the previously selected delay or falls back to the configured default.

Delay adaptation is explicitly rate-limited to avoid oscillation under noise, while step detection enables faster convergence after large workload transitions. All fitted coefficients are checked for finiteness before use; if fitting fails, the implementation

falls back to the previously valid parameter vector or a conservative default. Corrected emission is withheld entirely if these safeguards cannot be satisfied.

Implementation Consequences. During warmup, the raw Redfish integration remains the sole canonical system series. Once corrected reconstruction becomes ready, the implementation deletes the raw canonical series from the sink and emits only corrected values thereafter, preventing ambiguous double-publication under identical metric identifiers. The corrected counter does not reset at takeover due to the offset mechanism, but it remains a best-effort cumulative quantity that may restart on process restart. Overall, the implementation prioritises temporal consistency and conservatism: when alignment or anchoring is unreliable, raw integration remains authoritative.

Exported Metrics. The exported metrics are summarized in Table 5.12.

Metric name	Type	Unit	Labels
System metrics			
`system_power_mw`	Gauge	mW	`chassis, source, kind`
`system_energy_mj`	Counter	mJ	`chassis, source, kind`

Label domain: `kind` = total, `source` ∈ {`redfish_raw`, `redfish_corrected`}.

TABLE 5.12: Exported Redfish-derived system metrics.

5.7.9 Stage 2: System-Level Energy Model and Residual

Architectural Context. This stage realizes the system-level energy balance defined in § 4.8.3. Its role is to operationalize residual energy as a conservative, node-local quantity derived from system and component observations, while tolerating asynchronous and delayed inputs. No workload attribution or semantic decomposition is performed at this stage.

Implementation Strategy. For each analysis window, the implementation derives window energy from window-averaged power and window duration for both system-level and component-level signals. Residual energy is computed as a window-local difference and accumulated into a monotonic counter. Power-level residual metrics are treated as auxiliary observables, whereas cumulative energy counters are considered authoritative. An explicit window validity signal is emitted to indicate when residual interpretation is temporally reliable.

Core Mechanisms. Residual energy accumulation follows the architectural definition of E_{res} from § 4.8.3. Window energy increments are clamped to non-negative values before accumulation to preserve monotonicity of the exported counter. This clamping is applied strictly at the accumulation boundary and does not alter the conceptual definition of residual energy. Residual power is derived from the same window-local quantities but is not used as a basis for accumulation.

Robustness and Edge Cases. Temporal misalignment between system-level and component-level signals may lead to transient inconsistencies at the power level.

These effects are contained by decoupling instantaneous power from cumulative energy accounting. Rather than masking such artifacts, the implementation exposes an explicit window usability signal, allowing downstream stages to reason about residual validity without compromising conservation. Warmup behavior is handled by maintaining continuity between raw and corrected system sources while preserving counter monotonicity.

Implementation Consequences. The implementation guarantees strict energy conservation at both window and cumulative levels. Residual energy counters remain monotonic and conservative under all operating conditions. Transient timing artifacts originating from system-level measurement latency may appear in auxiliary power metrics but do not affect energy correctness. Residual energy thus forms a stable and auditable constraint for downstream decomposition and attribution stages.

Exported Metrics. The exported metrics are summarized in Table 5.13.

Metric name	Type	Unit	Labels
Residual energy and power			
`residual_energy_mj`	Counter	mJ	`chassis, source, kind`
`residual_power_mw`	Gauge	mW	`chassis, source, kind`
Residual window validity			
`residual_window_usable`	Gauge	bool	`chassis, source`

Label domain: `kind` = total.

TABLE 5.13: Exported residual metrics.

5.7.10 Stage 3: Idle and Dynamic Energy Semantics

5.7.10.1 RAPL Idle and Dynamic Decomposition

Architectural Context. This section realises the architectural idle and dynamic split defined in § 4.8.4.1 for RAPL domains by estimating a conservative idle baseline from utilization-conditioned observations and materialising a per-window energy decomposition that preserves strict conservation.

Implementation Strategy. The implementation derives idle power from the already exported total RAPL energy by differencing cumulative counters per window and estimating a baseline power β_d from low-utilization operating points. Utilization proxies are obtained directly from raw eBPF process ticks within the effective window to avoid dependencies on intermediate metric availability. Idle estimation and smoothing are maintained as explicit cross-window state, while the per-window split is computed deterministically from the current baseline and total energy increment.

Core Mechanisms. For each domain, the cumulative total energy is converted to a window increment by differencing against the last observed value and clamping negative deltas to zero. A utilization proxy u_d is computed by normalizing the window rate of eBPF-derived activity against a rolling p95 amplitude, yielding a

bounded proxy in $[0, 1]$. Pairs (u_d, P_d^{tot}) are fed into a scalar idle model that performs bucketed regression over low-utilization samples and returns the intercept as the candidate idle baseline. Baseline updates are asymmetrically smoothed in time, permitting faster decreases than increases and suppressing upward adaptation until the p95 normalization is deemed stable. The resulting idle power is clamped to the observed total power for the same window, converted to an idle energy increment, and subtracted from the total increment to obtain the dynamic remainder. Both increments are accumulated into monotonic per-domain counters and exposed together with window-average powers.

Robustness and Edge Cases. If total energy is unavailable or observed for the first time, the cycle is skipped to avoid poisoning state. Idle power is constrained to be non-negative and never exceed total power, ensuring that the split cannot generate energy even under transient noise or model error. When utilization normalization is not yet stable, baseline increases are disabled to prevent premature upward bias, while decreases remain bounded to resist single-sample outliers. All state required for differencing, normalization, modeling, and smoothing is scoped per domain, preventing cross-domain interference.

Implementation Consequences. The realised split guarantees per-window energy conservation and produces a conservative idle estimate that converges under sustained low activity while remaining safe on permanently loaded systems. Dynamic energy is defined purely as the residual, ensuring that subsequent attribution stages operate on a well-bounded remainder without requiring access to the idle model internals. The approach tolerates missing or delayed observations and degrades to zero-idle attribution rather than emitting inconsistent metrics.

While the CPU attribution path is comparatively compact, this reflects the availability of continuous, architecturally exposed counters and execution-derived proxies; GPU attribution necessarily incurs additional complexity due to backend-controlled publication semantics, implicit sampling phases, and heterogeneous telemetry guarantees.

Exported Metrics. The exported metrics are summarized in Table 5.14.

Metric name	Type	Unit	Labels
Idle and dynamic RAPL metrics			
`rapl_energy_mj`	Counter	mJ	`domain, kind, source`
`rapl_power_mw`	Gauge	mW	`domain, kind, source`

Label domain: `domain` $\in \{\text{pkg}, \text{core}, \text{uncore}, \text{dram}\}$, `kind` $\in \{\text{idle}, \text{dynamic}\}$, `source` $=$ `rapl`.

TABLE 5.14: Exported RAPL idle and dynamic metrics.

5.7.10.2 Residual Idle and Dynamic Decomposition

Architectural Context. This subsubsection realises the residual idle and dynamic decomposition defined in the architecture, which introduces a persistent residual idle baseline and a conditional interpretation contract based on window usability. The implementation enforces exact conservation on the clamped residual budget

while ensuring that learning and interpretation are explicitly gated under temporal misalignment.

Implementation Strategy. Residual idle and dynamic power are constructed from a non-negative residual budget derived per analysis window. A persistent idle baseline is maintained as explicit cross-window state and is reused whenever learning is not permitted. Learning of this baseline is strictly conditional on corrected system input, window usability, and a minimum residual magnitude. Metric emission remains continuous and conservative in all windows, while baseline updates are selectively enabled.

Core Mechanisms. For each window, residual total power is obtained by subtracting the sum of aligned component power from system power and clamping the result to a non-negative budget. Temporal misalignment is detected using conservative window-local indicators, including sustained lag of system power behind accounted components and sharp increases in component power coinciding with near-zero or clamped residuals. A short hold horizon is applied to transient classification to prevent flapping. Window usability is defined as the conjunction of non-transient classification and the absence of residual clamping. The residual idle baseline is seeded on the first window that satisfies all learning preconditions. Subsequent updates follow a candidate-then-commit mechanism that tracks sustained low residual observations and applies bounded downward adjustments only after confirmation. Idle and dynamic residual power are materialized conservatively by capping idle at the clamped residual budget and assigning the remainder to dynamic, ensuring exact per-window conservation. Energy counters are obtained by integrating window-local power and accumulated monotonically using the clamped budget, with explicit handling of the raw-to-corrected system metric transition.

Robustness and Edge Cases. Unusable windows do not trigger idle baseline learning and cannot decrease the baseline, preventing collapse during temporal misalignment. Learning is forbidden during warmup to avoid contamination from raw system metrics. Residual budgets below a minimum magnitude are excluded from learning to prevent noise fitting. Downward baseline adaptation is hardened by requiring persistence across multiple consecutive learnable windows and by rate-limiting the maximum permanent drop per commit. The idle component is always capped by the current clamped residual budget, preventing negative dynamic residuals even when the baseline temporarily exceeds the budget. Once corrected system metrics become active, raw residual series are explicitly removed to avoid overlapping series with incompatible semantics.

Implementation Consequences. Residual idle and dynamic metrics are conservative by construction and conserve the clamped residual budget exactly in every window. Persistent idle state remains stable under aggressive workload changes because temporally invalid windows are excluded from learning and the last committed baseline is held constant. The usability signal establishes a hard contract for downstream consumers: residual idle and dynamic values may be present in all windows, but are only interpretable and learnable when usability is true. The exported idle baseline is explicit model state and must not be interpreted as a physical idle quantity.

Exported Metrics. The exported metrics are summarized in Table 5.15.

Metric name	Type	Unit	Labels
Residual idle and dynamic metrics			
`residual_power_mw`	Gauge	mW	`chassis, source, kind`
`residual_energy_mj`	Counter	mJ	`chassis, source, kind`
Residual idle model state			
`residual_idle_baseline_mw`	Gauge	mW	`chassis, source`
Residual window validity			
`residual_window_usable`	Gauge	bool	`chassis, source`

Label domain: `kind` $\in$ {idle, dynamic}, `source` $\in$ {`redfish_raw`, `redfish_corrected`}.

TABLE 5.15: Exported residual idle and dynamic metrics.

5.7.10.3 GPU Idle and Dynamic Decomposition

Architectural Context. This section realizes the architectural GPU idle and dynamic decomposition defined in § 4.8.4.3, operating exclusively on the corrected per-device total power and window-consistent duration.

Implementation Strategy. For each GPU device, the implementation maintains a per-device idle estimator whose input is the corrected total power together with a compact utilization signal. Idle estimation is updated opportunistically under stable conditions and otherwise held constant, ensuring that transient load changes do not perturb the baseline. Idle and dynamic components are derived per window from the same total power observation and duration, then accumulated into monotonic energy counters.

Core Mechanisms. The implementation retrieves the corrected total power gauge and the corresponding window duration, and maps the most recent device utilization observation into the corrected window. An idle estimator is instantiated per device and observes tuples $(u_{\text{sm}}, u_{\text{mem}}, P_{\text{total}})$. A new idle estimate is accepted only when utilization remains approximately constant and within a low-utilization region, yielding a stable baseline β_{uuid}. For each window, idle power is obtained by clamping β_{uuid} to the current total power, dynamic power is computed as the non-negative residual, and both are converted to window energy increments using the same duration as the total. All three cumulative energies (total, idle, dynamic) are updated atomically in per-device state.

Robustness and Edge Cases. If total power is unavailable for a device in a window, no idle or dynamic update is performed to avoid inconsistent state. Idle estimates are bounded to the interval $[0, P_{\text{total}}]$ to prevent negative or superlinear components. Stability gating prevents sudden utilization changes from contaminating the idle baseline, causing such transients to be attributed entirely to dynamic power. All intermediate values are checked for non-finite results and clipped to preserve monotonicity of cumulative energy counters.

Implementation Consequences. The implementation enforces a conservative decomposition in which idle power evolves slowly and only under stationary conditions, while dynamic power absorbs short-term fluctuations. Energy conservation holds per window by construction, and cumulative counters remain monotonic over the lifetime of the system. The per-device state isolation allows heterogeneous GPUs to be handled independently without cross-coupling effects.

Exported Metrics. The exported metrics are summarized in Table 5.16.

Metric name	Type	Unit	Labels
GPU idle and dynamic decomposition metrics			
`gpu_power_mw`	Gauge	mW	`gpu_uuid, kind, source`
`gpu_energy_mj`	Counter	mJ	`gpu_uuid, kind, source`

Label domain: `kind` $\in$ {idle, dynamic}, `source` = `nvml_corrected`.

TABLE 5.16: Exported GPU idle and dynamic decomposition metrics.

5.7.11 Stage 4: Workload Attribution and Aggregation

5.7.11.1 Workload Attribution of eBPF Utilization Counters

Architectural Context. This section realizes the workload-level aggregation model, mapping per-process eBPF utilization deltas to workload-attributed monotonic counters while enforcing strict conservation and residual `__system__` semantics.

Implementation Strategy. Attribution is implemented as a window-local aggregation over the effective eBPF observation window. All raw per-process deltas observed in the window are treated as authoritative inputs. Resolution to workload identity is attempted per process using the standard Stage 4 resolver, but aggregation into non-system workloads is conditional. Unresolved activity is not dropped or heuristically reassigned and is instead captured explicitly via residual construction.

Core Mechanisms. For each effective window, per-process deltas are integrated over the window interval using fractional overlap factors to account for partial boundary intersections. Two accumulations are performed in parallel. First, signal-wise totals are computed by summing all observed per-process deltas, independent of resolution. Second, resolved per-process deltas are aggregated into per-workload buckets. After aggregation, the `__system__` workload is materialized as the residual between the total activity and the sum of all non-system workloads, with negative residuals clamped to zero to preserve non-negativity. All workload aggregates are accumulated into state-backed monotonic counters, preserving counter semantics across windows.

Robustness and Edge Cases. Partial observability affects only resolution, not conservation. If process identity or workload metadata is unavailable, unstable, or inconsistent, the corresponding deltas contribute solely to the total and therefore increase the `__system__` residual. Window boundary truncation is handled via fractional scaling, with accumulators maintained in floating-point state to avoid systematic loss under repeated truncation. Emitted counter values are monotonically

non-decreasing by construction, and defensive checks prevent regression under any execution order. Workload series are garbage-collected after a bounded inactivity period, while the `__system__` series is explicitly exempt from deletion.

Implementation Consequences. The implementation guarantees strict conservation between system-level eBPF counters and the sum of workload-attributed counters over any analysis horizon. Resolution failure degrades monotonically into `__system__` without redistributing activity among resolved workloads. The resulting counters are suitable as first-class workload attribution outputs and as explanatory inputs for subsequent energy attribution and validation stages.

Exported Metrics. The exported metrics are summarized in Table 5.17.

Metric name	Type	Unit	Labels
CPU execution activity			
`workload_bpf_cpu_instructions_total`	Counter	count	*common (see below)*
`workload_bpf_cpu_cycles_total`	Counter	count	*common (see below)*
`workload_bpf_cache_misses_total`	Counter	count	*common (see below)*
`workload_bpf_page_cache_hits_total`	Counter	count	*common (see below)*
Interrupt activity			
`workload_bpf_irq_net_tx_total`	Counter	count	*common (see below)*
`workload_bpf_irq_net_rx_total`	Counter	count	*common (see below)*
`workload_bpf_irq_block_total`	Counter	count	*common (see below)*
Process runtime			
`workload_bpf_process_run_us_total`	Counter	μs	*common (see below)*

Label domain: `source` = `bpf`; labels `namespace`, `pod`, and `container` identify the workload.

TABLE 5.17: Exported workload-attributed eBPF utilization counters.

5.7.11.2 Workload Resolution and Identity Enforcement

Architectural Context. This section realizes the workload resolution abstraction introduced in the Stage 4 architecture. The resolver provides a conditional mapping from execution-level observations to Kubernetes workload identities and enforces the semantics of the distinguished `__system__` class. It is an auxiliary component of the attribution pipeline: attribution logic consumes its outputs but does not embed or compensate for resolution failures. All guarantees are enforced here under discretization, metadata delay, and partial observability.

Implementation Strategy. Workload resolution is implemented as a best-effort, cycle-scoped join between execution identities and the metadata store associated with the current analysis cycle. Resolution is intentionally asymmetric: positive identification requires multiple consistency checks, while failure at any point causes immediate fallback to an unresolved state. The resolver never invents workload identities, never extrapolates beyond available metadata, and never retries resolution retroactively across cycles. This strategy ensures conservative behavior and monotone degradation toward `__system__` under uncertainty.

Core Mechanisms. Resolution proceeds in three ordered stages. First, a process-based path attempts to resolve a stable process identity using a guarded `(PID, StartJiffies)` pair to harden against PID reuse. Second, when process identity is unavailable or unsafe, a cgroup-based fallback is attempted using only explicitly attributable cgroup identifiers. Third, a resolved container identifier is mapped to Kubernetes namespace, pod, and container labels via the metadata store. Only when all required identity components are present is a workload key materialized; otherwise resolution fails. The resolver itself returns either a fully specified workload key or an unresolved outcome, with no intermediate states.

Robustness and Edge Cases. All resolution steps are guarded against known failure modes. PID reuse is explicitly checked and causes conservative rejection rather than reassignment. Sentinel or root cgroup identifiers are excluded to prevent poisoning of the mapping chain. Missing, stale, or incomplete metadata entries immediately invalidate resolution. Importantly, resolution failure does not propagate partial identity information: unresolved observations are not weakly labeled but are handled uniformly by the attribution layer as `__system__`. This guarantees that loss of metadata cannot redistribute energy between resolved workloads.

Implementation Consequences. The resolver enforces a strict correctness boundary for Stage 4 attribution. When identity information is reliable, workload attribution is precise and reproducible within the limits of the chosen fairness basis. When identity information degrades, attribution degrades conservatively and explicitly without violating conservation or introducing hidden assumptions. This behavior ensures that workload-resolved metrics remain interpretable across runs with varying metadata quality and system churn.

Exported Metrics. The workload resolver does not export metrics directly. Its effects are observable only through the workload-labeled metrics produced by subsequent attribution stages, including the explicit appearance of the `__system__` workload class when resolution is not possible.

5.7.11.3 CPU Dynamic Energy Attribution

Architectural Context. This section realizes the bin-level CPU dynamic attribution model defined in the corresponding architecture section. For each RAPL CPU domain, a per-window dynamic energy budget is distributed across workloads using execution-derived activity proxies, while enforcing conservation, non-negativity, completeness, and monotone degradation toward `__system__` under partial observability.

Implementation Strategy. The implementation treats each RAPL domain uniformly and executes attribution independently per domain. The authoritative input is the per-window delta of the cumulative dynamic energy counter for the domain. Workload attribution is computed at native eBPF tick resolution and only aggregated to window scope after all bin-level decisions have been made. All attribution state is explicit and window-scoped, with no cross-window reinterpretation.

Core Mechanisms. Per-bin activity weights are constructed from eBPF process counters by differencing against state-managed baselines. Baselines are keyed by

a stable process identity when available and fall back conservatively when not. For each bin, total activity mass and per-workload activity mass are accumulated according to the domain-specific proxy. The window-level dynamic energy budget is then allocated across bins proportionally to their activity mass, and each bin's energy share is further allocated across workloads proportionally to their per-bin activity. Exact conservation in integer millijoules is enforced using a largest-remainder finalization step, with any rounding remainder routed explicitly to `__system__`.

Robustness and Edge Cases. Missing or zero activity mass is handled conservatively. If no activity is observed for a domain across the entire window, the full dynamic energy budget is assigned to `__system__`. If activity is absent in an individual bin, that bin's energy share is assigned to `__system__`. For the `dram` domain, a window-scoped fallback replaces cache-miss activity with CPU cycle activity when cache-miss mass is zero but cycle mass is non-zero. Counter resets, PID reuse, and partial process visibility are handled by baseline rebasing and by skipping unsafe deltas, which can only increase the `__system__` share and never redistribute energy among resolved workloads.

Implementation Consequences. The implementation guarantees conservative, monotonic, and temporally causal workload attribution of CPU dynamic energy. Integer rounding is resolved locally at both allocation stages using a largest-remainder scheme, ensuring exact conservation without cross-bin interference. Under reduced observability, attribution degrades only by increasing the `__system__` share, without redistributing energy among resolved workloads. The explicit bin-level realization preserves the architectural contract and enables later extensions such as idle allocation and residual energy handling without modifying the dynamic attribution core.

Exported Metrics. The exported metrics are summarized in Table 5.18.

Metric name	Type	Unit	Labels
CPU dynamic workload energy			
`workload_rapl_energy_mj`	Counter	mJ	*common (see below)*

Label domain: `domain` $\in$ {pkg, core, uncore, dram}, `kind` = dynamic, `source` = `rapl`; labels `namespace`, `pod`, and `container` identify the attributed workload.

TABLE 5.18: Exported CPU dynamic workload energy metrics.

5.7.11.4 CPU Idle Energy Attribution

Architectural Context. This section realizes the CPU idle attribution model defined in the corresponding architecture subsubsection, enforcing a two-pool fairness policy under discretized, window-based execution. The implementation adheres to the Stage 4 invariants and mirrors the workload identity and lifecycle semantics used for CPU dynamic attribution.

Implementation Strategy. Per-window idle energy budgets are derived by differencing cumulative RAPL idle energy counters for each supported domain. Attribution is performed independently per domain, using domain-specific reservation signals while reusing the window-sticky activity weights produced by CPU dynamic

attribution. All allocation decisions are local to the current window and do not depend on future information.

Core Mechanisms. For each domain, the idle budget is split into a reserved and an opportunistic pool using the domain-specific reservation fraction β. Reservation weights are obtained from container-level CPU requests for `pkg`, `core`, and `uncore`, and from container-level memory requests for `dram`. Activity weights are taken from the window-aggregated dynamic allocation maps.

The eligible workload set is defined as the union of workloads observed in dynamic attribution, currently running containers, and the mandatory `__system__` workload. The reserved pool is allocated proportionally among workloads with strictly positive requests. The opportunistic pool is allocated proportionally among workloads without requests and the `__system__` workload, using activity weights. Per-workload allocations from both pools are combined and accumulated into monotonic workload-level idle energy counters.

Robustness and Edge Cases. Missing or negative idle deltas result in zero budget for the affected window. If no valid requests exist, the reserved pool collapses to zero without redistributing energy. If no activity weights are available, the opportunistic pool collapses and the full idle budget is attributed to `__system__`. All rounding discrepancies are resolved deterministically, with any remainder routed to `__system__`, ensuring exact per-window conservation. Short-lived or metadata-late workloads may receive idle energy only via the opportunistic pool. Inactive workload series are garbage-collected using the same TTL-based mechanism as dynamic CPU attribution, without retroactive reinterpretation.

Implementation Consequences. The implementation guarantees strict conservation, non-negativity, and completeness for CPU idle energy attribution across all supported RAPL domains. Idle attribution degrades conservatively under partial observability and remains interpretable as a fairness policy rather than a causal signal. The resulting workload-level idle energy counters are directly comparable to dynamic CPU energy outputs and suitable for downstream aggregation and analysis.

Exported Metrics. The exported metrics are summarized in Table 5.19.

Metric name	Type	Unit	Labels
Workload-attributed CPU idle energy			
`workload_rapl_energy_mj`	Counter	mJ	*common (see below)*

Label domain: `domain` $\in$ {pkg, core, uncore, dram}, `kind` = idle, `source` = `rapl`; labels `namespace`, `pod`, and `container` identify the attributed workload.

TABLE 5.19: Exported CPU idle workload energy metrics.

5.7.11.5 GPU Dynamic Energy Attribution

Architectural Context. This subsection realises the Stage 4 GPU dynamic attribution rule by allocating corrected-series energy over process-sample intervals and aggregating by resolved workload identity, while routing unsupported fractions to `__system__`.

Implementation Strategy. For each GPU UUID, the implementation combines (i) a corrected energy series that can be integrated over arbitrary sub-intervals, (ii) a dynamic window budget, and (iii) a timestamped stream of per-process `ComputeUtil` snapshots. Attribution is performed by iterating sub-intervals induced by the snapshot times and applying a hold-last policy for utilization between snapshots.

Core Mechanisms. The corrected energy series is integrated over sub-intervals using zero-order hold on the fixed grid to obtain $\Delta E_{\text{gpu}}(I, \text{uuid})$. A per-window scale factor $f(W, \text{uuid})$ is computed from the ratio of the dynamic window budget to total corrected window energy and is clipped to $[0, 1]$. For each sub-interval, utilization mass is computed from the held PID map. If mass is zero, the entire sub-interval dynamic energy is accumulated to `__system__`. Otherwise, PID shares are computed proportionally and resolved to workloads using the existing resolver chain; unresolved shares are accumulated to `__system__`. Any floating-point remainder between interval budget and assigned shares is routed to `__system__` to preserve per-interval conservation.

Robustness and Edge Cases. Snapshot ingestion clamps `ComputeUtil` into $[0, 100]$ and treats invalid values as zero. If corrected-series integration fails for a sub-interval, no energy is allocated for that sub-interval, which preserves non-negativity and avoids inventing energy. If the corrected window energy is non-positive, or the corrected series is unavailable, the dynamic budget is conservatively routed to `__system__`. Resolver failure for a PID share does not affect other shares and degrades monotonically by redirecting only the unresolved share to `__system__`.

Implementation Consequences. Dynamic attribution preserves conservation against the constructed dynamic budget per GPU UUID and window, up to numerical round-off which is absorbed by `__system__`. Temporal allocation follows the observed snapshot cadence, preventing the coarse artifacts of reducing utilization to a single window statistic, while keeping the exported outputs window-scoped.

Exported Metrics. The exported metrics are summarized in Table 5.20.

Metric name	Type	Unit	Labels
Workload-attributed GPU dynamic energy			
`workload_gpu_energy_mj`	Counter	mJ	*common (see below)*

Label domain: `kind` = dynamic; labels `gpu_uuid`, `namespace`, `pod`, and `container` identify the attributed workload.

TABLE 5.20: Exported metrics for GPU dynamic workload attribution.

5.7.11.6 GPU Idle Energy Attribution

Architectural Context. This subsection realises the Stage 4 rule that GPU idle energy is not distributed to workloads and is instead emitted exclusively as `__system__`.

Implementation Strategy. The implementation mirrors the corrected idle GPU energy counter into the workload GPU energy metric for the `__system__` workload key, per GPU UUID.

Core Mechanisms. For each GPU UUID, the absolute corrected idle energy counter value is loaded and written into the corresponding `workload_gpu_energy_mj` series with `kind="idle"` and workload labels set to `__system__`. The write is monotonic-guarded to prevent counter regression if upstream inputs transiently decrease.

Robustness and Edge Cases. Invalid or negative idle counter values are clamped to zero. If the idle counter is missing for a device, no idle workload point is emitted for that device in the current cycle. Monotonic guarding ensures that transient upstream regressions do not violate the counter contract at the exporter boundary.

Implementation Consequences. Idle workload energy is complete by construction and exactly consistent with corrected idle energy, while avoiding speculative distribution. Together with dynamic attribution, the workload GPU energy metric admits a complete accounting view in which idle energy is explicitly represented as `__system__`.

Exported Metrics. The exported metrics are summarized in Table 5.21.

Metric name	Type	Unit	Labels
Workload-attributed GPU idle energy			
`workload_gpu_energy_mj`	Counter	mJ	*common (see below)*

Label domain: `kind` = idle; labels `gpu_uuid`, `namespace`, `pod`, and `container` identify the attributed workload.

TABLE 5.21: Exported metrics for GPU idle workload attribution.

5.7.12 Prometheus Exporter Implementation

Architectural Context. This implementation realizes the passive exposition model defined above by translating committed analysis points into a Prometheus-compatible pull interface.

Implementation Strategy. Analysis points are pushed into the exporter as immutable values and retained only as the most recent sample per metric series. Metric families are created lazily on first observation, with a fixed label schema determined at discovery time.

Core Mechanisms. Each analysis metric identifier is mapped to a sanitized Prometheus metric name with a mandatory "`tycho_`" prefix. Label keys are fixed on first sight and reused for all subsequent emissions. At scrape time, the exporter exposes the latest committed value for each active series without additional buffering or recomputation.

Robustness and Edge Cases. Schema instability is handled conservatively by ignoring newly appearing labels after first observation. Exporter startup is decoupled from analysis initialization, ensuring availability even while upstream components perform long-running calibration or setup. Multiple sinks may coexist, allowing Prometheus export and auxiliary sinks such as logging to operate concurrently.

Implementation Consequences. The exporter preserves analysis correctness by construction, as it neither modifies nor reinterprets metric values. All temporal guarantees are inherited directly from the analysis engine, while exposition remains strictly observational and replaceable. By remaining strictly passive and state-free with respect to attribution, the exporter preserves the analysis invariants of determinism, non-retrospective execution, and window-scoped finality established throughout the implementation.

5.8 Summary

This chapter described how Tycho's architectural abstractions are realized as a deterministic, execution-time system under discretization, partial observability, and asynchronous measurement. The implementation enforces correctness properties through execution structure rather than adaptive control, ensuring that attribution semantics are explicit, auditable, and reproducible.

At runtime, Tycho is organized as a set of long-lived subsystems with strictly separated responsibilities. Temporal coordination, observation, identity acquisition, calibration, analysis, and export are decoupled by construction, and no subsystem compensates implicitly for the behavior of another. A global monotonic timebase, explicit analysis windows, and single-pass execution ensure that each analysis cycle yields a final, window-scoped interpretation of the available evidence.

Metric construction and attribution are realized as a staged pipeline. Early stages construct component-level energy and utilization signals while preserving monotonicity and conservation. Subsequent stages decompose these signals into idle and dynamic components and allocate dynamic energy to workloads using execution-derived proxies. Workload resolution is conservative and explicitly bounded, with unresolved activity degrading monotonically into the distinguished `__system__` class rather than being redistributed.

Across all stages, the implementation prioritizes conservative behavior under uncertainty. Missing or delayed observations do not trigger backfilling or reinterpretation of prior windows. Instead, degradation is explicit, localized, and bounded, preserving the interpretability of all emitted metrics. Cumulative energy counters remain monotonic, per-window conservation is enforced exactly, and all stateful behavior is confined to explicit, metric-owned memory.

Together, these implementation choices realize the architectural model faithfully while remaining robust under real execution conditions. They establish a stable and well-defined foundation for the evaluation of measurement accuracy, attribution behavior, and system overhead in the following chapters.

Appendix A

Container-Level Energy Consumption Estimation: Foundations, Challenges, and Current Approaches

Zurich University of Applied Sciences

Department School of Engineering

Institute of Computer Science

SPECIALIZATION PROJECT 2

Container-Level Energy Consumption Estimation: Foundations, Challenges, and Current Approaches

---

Author:
Caspar Wackerle

Supervisors:
Prof. Dr. Thomas Bohnert
Christof Marti

Submitted on
JULY 31, 2025

Re-edited on
January 31, 2026

Study program:
Computer Science, M.Sc.

Abstract

The growing energy demands of data centers have positioned energy efficiency as a critical concern in modern cloud computing. As containerization becomes the dominant approach for deploying scalable workloads, understanding the energy consumption of individual containers gains strategic relevance. However, accurately attributing energy usage to containerized workloads remains a complex and largely unsolved challenge due to hardware abstraction, shared resource utilization, and limited telemetry visibility.

This thesis investigates the theoretical foundations, methodological challenges, and existing approaches to container-level energy consumption measurement. Emphasizing bare-metal Kubernetes environments, the study systematically explores system-level energy measurement techniques, the complexities of attributing node-level energy to individual containers, and the limitations of current measurement tools. Rather than developing a new estimation tool, this work provides a structured analysis of existing solutions, highlighting methodological gaps, validation challenges, and critical design considerations for future research and tool development.

The findings offer a consolidated understanding of the technical factors influencing container energy attribution and outline practical recommendations for advancing energy transparency in containerized cloud infrastructures.

Chapter 1

Introduction

1.1 Cloud Computing and its Impact on the Global Energy Challenge

Global energy consumption is rising at an alarming pace, driven in part by the accelerating digital transformation of society. A significant share of this growth comes from data centers, which form the physical backbone of cloud computing. While the cloud offers substantial efficiency gains through resource sharing and dynamic scaling, its aggregate energy footprint is growing rapidly. Data centers accounted for around 1.5% (approximately 415 TWh) of the world's electricity consumption in 2024 and are set to more than double by 2030[56]. This figure slightly exceeds Japan's current electricity consumption.

This increase is fueled by the rising demand for compute-heavy workloads such as artificial intelligence, large-scale data processing, and real-time services. Meanwhile, traditional drivers of efficiency (such as Moore's law and Dennard scaling) are slowing down[57, 58]. Improvements in data center infrastructure, like cooling and power delivery, have helped reduce energy intensity per operation[59], but these gains are approaching diminishing returns. As a result, total data center energy use is expected to grow faster than before, as efficiency per unit of compute continues to improve more slowly[60]. As containerized workloads form a significant and growing fraction of data center operations, understanding their energy impact is of increasing relevance.

1.1.1 Rise of the Container

Containers have become a core abstraction in modern computing, enabling lightweight, fast, and scalable deployment of applications. Compared to virtual machines, containers impose less overhead, start faster, and support finer-grained resource control. As such, they are widely used in microservice architectures and cloud-native environments[61].

This trend is amplified by the growing popularity of Container-as-a-Service (CaaS) platforms, where containerized workloads are scheduled and managed at high density on shared infrastructure. Kubernetes has become the de facto orchestration tool for managing such workloads at scale. While containers are inherently more energy-efficient than virtual machines in many scenarios[62], their widespread use introduces a critical complication: accurately understanding and attributing their energy

consumption. Despite their operational advantages, containers obscure energy usage due to their shared-resource architecture, making transparent monitoring and assessment of their true energy efficiency significantly more difficult.

1.1.2 Thesis Context and Motivation

This thesis is part of the Master's program in Computer Science at the Zurich University of Applied Sciences (ZHAW) and represents the second of two specialization projects ("VTs"). The preceding project (VT1) focused on the practical implementation of a test environment for energy efficiency research in Kubernetes clusters. This thesis (VT2) is intended to explore the theoretical and methodological aspects of container energy consumption measurements in detail.

Furthermore, this thesis builds upon prior works focused on performance optimization and energy measurement. EVA1 covered topics such as operating system tools, statistics, and eBPF, while EVA2 explored energy measurement in computer systems, covering energy measurement in computer systems at the hardware, firmware, and software levels. This thesis builds upon these foundations, focusing specifically on the problem of container-level energy consumption measurement.

1.1.3 Use of AI Tools

During the writing of this thesis, *ChatGPT*[63] (Version 4o, OpenAI, 2025) was used as an auxiliary tool to enhance efficiency in documentation and technical writing. Specifically, it assisted in:

- Assisting in LaTeX syntax corrections and document formatting.
- Improving clarity and structure in selected technical explanations.
- Supporting minor code analysis and debugging tasks.

All AI-generated content was critically reviewed, edited, and adapted to fit the specific context of this thesis. **AI was not used for literature research, conceptual development, methodology design, or analytical reasoning.** The core ideas, analysis, and implementation details were developed independently.

1.1.4 Container Energy Consumption Measurement Challenges

Containerized environments introduce fundamental challenges to energy consumption measurement. Unlike virtual machines, containers share the host operating system kernel and underlying hardware resources. This shared architecture obscures the direct relationship between a specific container and the physical energy consumed, making isolated measurement infeasible.

While modern processors expose hardware-level telemetry via interfaces such as Intel's Running Average Power Limit (RAPL), these provide only node-level or component-level insights. In addition, such telemetry is typically inaccessible from within containers, particularly in multi-tenant or cloud-hosted environments. Public cloud providers often aggregate or abstract energy-related data, further limiting observability.

Attribution of energy consumption to containers is complicated by concurrent resource sharing. CPU cores, memory, network interfaces, and storage systems serve multiple containers simultaneously. Available resource utilization metrics, such as CPU time, memory usage, or performance counters, provide indirect signals that could be used for energy attribution, but don't directly attribute it.

Various tools attempt to model container-level power consumption by correlating resource utilization with node-level energy telemetry. However, these models are often simplistic, opaque, or specific to particular hardware setups, and lack systematic validation.

Collectively, these factors result in a complex, multi-layered measurement problem: translating node-level energy consumption into accurate, container-level usage estimates within modern cloud infrastructures.

1.1.5 Scope and Research Questions

This thesis investigates the theoretical and practical landscape of container energy consumption measurement, focusing on the attribution of energy usage within bare-metal Kubernetes environments. Instead of proposing a novel measurement tool or developing a new energy estimation model, the thesis aims to systematically analyze the problem space and existing solutions.

The objective is to identify the methodological, technical, and practical factors that influence accurate container-level energy measurement. The study evaluates how existing tools address these challenges and highlights unresolved issues that hinder reliable and standardized energy attribution in containerized systems.

To guide this exploration, the following research questions are posed:

- **RQ1:** What are the fundamental challenges that prevent accurate measurement of container-level energy consumption?
- **RQ2:** Which methods, metrics, and models currently support container energy consumption estimation, and how do they address the attribution problem?
- **RQ3:** How do existing tools implement container-level energy consumption estimation, and what are the limitations of their approaches?

Rather than explicitly answering these questions in isolation, the thesis addresses them throughout its analysis: These questions structure the thesis' analytical approach. A detailed overview of the thesis structure is provided in the following section.

1.1.6 Terminology: Power and Energy

In this thesis, the physical units power (measured in watts) and energy (measured in joules) are used interchangeably where appropriate. While these units describe distinct physical quantities (energy representing the total amount of work performed, and power representing the rate of energy usage over time), the time interval in question is generally known or defined in all relevant contexts, rendering conversion between them trivial.

As a result, discussions of container-level energy consumption, power usage, and energy attribution may reference either energy or power depending on context, without introducing ambiguity. Where necessary, the specific unit used is stated explicitly.

1.1.7 Contribution and Structure of the Thesis

This thesis contributes a structured, theory-focused exploration of container-level energy consumption measurement. Rather than presenting a novel tool or proposing a specific implementation concept, it synthesizes existing methods, models, and challenges relevant to this field. Its primary contribution lies in analyzing how energy consumption can be measured and attributed to individual containers in bare-metal Kubernetes environments, highlighting limitations, and identifying open challenges for future work.

The thesis is structured as follows:

- **Chapter 2** introduces the fundamentals of server energy consumption, including hardware-level telemetry, component-level energy behaviors, and system power management mechanisms.
- **Chapter 3** analyzes the attribution problem of mapping node-level energy consumption to individual container workloads, identifying core challenges and influencing factors.
- **Chapter 4** surveys existing tools and approaches that attempt to estimate container-level energy consumption, assessing their methodologies and limitations.
- **Chapter 5** synthesizes the findings, identifies open questions, and outlines recommendations for future research and tool development.

Both this thesis and the preceding implementation-focused project are publicly available in the PowerStack[1] repository on GitHub. While this thesis does not come with additional code, the repository contains Ansible playbooks for automated deployment, Kubernetes configurations, monitoring stack setups, and benchmarking scripts from the preceding thesis.

Chapter 2

State of the Art and Related Research

2.1 Energy Consumption Measurement and Efficiency on Data Center Level

Energy consumption and efficiency at the data center level have been well-studied to the point where various literature reviews have been published[3, 64]. Much of this research focuses on data center infrastructure (cooling and power), and with good reason, as infrastructure is responsible for a large share of total energy consumption. While a wide range of coarse-, medium-, and fine-grained metrics for data center energy consumption exist, most operators have concentrated on improving coarse-grained metrics, especially *Power Utilization Effectiveness (PUE)*, through infrastructure improvements. This has resulted in a PUE of 1.1 or lower in some cases[59]. Meanwhile, server energy efficiency has improved substantially, particularly for partial load and idle power[65]. This has allowed data center operators to improve energy efficiency simply by installing more efficient cooling and power systems and servers. Fine-grained metrics, such as server component utilization rates or speeds, were generally not used in the context of energy efficiency but rather as performance metrics to ensure customer satisfaction.

2.2 Energy Consumption Measurement on Server Level

As a result of the energy efficiency improvements of both data center infrastructure and server hardware mentioned in the previous section, attention has shifted towards evaluating actual server load energy efficiency. Efficiency gains at this level compound into further improvements at the data center level. The resource-sharing methods of modern cloud computing(and especially the use of containers)create significant opportunities for server workload optimization for energy efficiency, which in turn requires power consumption measurements for evaluation. In the context of containers on multi-core processors, measuring the energy consumption of the entire server is insufficient, as it does not allow attribution of consumed energy to specific containers or processes. While component-level power measurements provide finer granularity and could theoretically be modeled to reflect container energy consumption, they drastically raise complexity for several reasons:

- Component-level energy consumption measurement without external tools is

far from straightforward. While some components provide estimation models (e.g. Intel RAPL or *Nvidia Management Library* (NVML)), others can only be estimated using performance metrics. This invariably leads to large measurement uncertainties, especially due to hardware differences between generations and manufacturers.

- Attributing measured or estimated energy consumption to individual containers is itself a non-trivial problem: it requires fine-grained time synchronization between energy consumption and container resource usage due to the fast-switching nature of most server components during multitasking.

- A deep understanding of dynamic versus static energy consumption is required: depending on the attribution model, a container might account not only for the energy it actively used but also for a fraction of the energy consumed by shared overhead, such as shared hardware components or system resources (such as the Kubernetes system architecture). This idea can be extended further: containers could potentially be penalized for unused server resources, as unused capacity still consumes energy. These different attribution models lead to a broader debate about the goals of the measurements.

- Any server-level power models used to estimate the relationship between individual component energy consumption suffer from the variety of server configurations, due to specialization such as storage-, GPU-, or memory-optimized servers.

In a systematic review of cloud server power models, Lin et al.[2] categorize power collection methods into four categories:

Key	**Value**	**Description**	**Deployment Difficulty**	**Data Granularity**	**Data Credibility**
Based on instruments	Installation of extra devices	Bare-metal machines	Easy	Machine Level	Very high
Based on dedicated acquisition system	Specialized systems	Specified models of machines	Difficult	Machine or component-level	High
Based on software monitoring	Built-in power models	Bare-metal and virtual servers	Moderate	Machine, component, or VM level	Fair
Based on simulation	System simulation	Machine, component, or VM level	Easy	Machine, component, or VM level	Low

TABLE 2.1: Comparison of power collection methods for cloud servers

The following sections of this chapter aim to present the current state-of-the-art in the various fields of research related to the problem domains listed above, focusing on different measurement approaches: direct hardware measurements, in-band measurement techniques, and model-based estimation. The sections are organized by measurement approach, foregoing organization by server component. For this reason, § 2.7 provides a brief summary of component-specific energy consumption

measurement techniques.

2.3 Direct Hardware Measurement

2.3.1 Instrument-based Power Data Acquisition

Instrument-based data acquisition produces the highest data credibility at low granularity: these devices, installed externally (measuring the power supplied to the PDU) or internally (measuring the power flow between the PDU and motherboard), have been the source of information for a number of studies. The approach of simply measuring electric power at convenient hardware locations using dedicated equipment can, of course, be extended to provide additional granularity. For example, Desrocher et al.[8] custom-created a DIMM extender fitted with Hall-sensor resistors and a Linux measurement utility to measure the power consumed by a DIMM memory module at a 1kHz sampling rate, using a *WattsAppPro?* power meter and a *Measurement Computing USB.1208FS-Plus* data acquisition board.

This highlights a fundamental truth of instrument-based data collection: while it is possible to implement a measuring solution that provides high-granularity and high-sampling rate power data, this requires immense effort, as such solutions are not available off-the-shelf. Unsurprisingly, this is most valuable for benchmarking or validation (Desrochers et al. used their setup to validate Intel RAPL DRAM power estimations on three different systems). However, this methodology is currently unsuitable for deployment in data center servers due to its poor scalability and prohibitive costs. Hence, the primary role of instrument-based power data acquisition is as a benchmarking and validation tool for research and development.

2.3.2 Dedicated Acquisition Systems

2.3.2.1 BMC Devices, IPMI and Redfish

Some manufacturers have developed specialized power data acquisition systems for their own server products. The baseboard management controller (BMC) is a typical dedicated acquisition system usually integrated with the motherboard, typically as part of the Intelligent Platform Management Interface (IPMI)[2]. It can be connected to the system bus, sensors, and a number of components to provide power and temperature information about the CPU, memory, LAN port, fan, and the BMC itself. Some comprehensive management systems, such as Dell iDRAC or Lenovo xClarity, have been further developed to provide high-quality, fine-grained power data due to their close integration between system software and underlying hardware. BMC devices on modern servers often offer IPMI or Redfish interfaces. While these interfaces use the same physical sensors, their implementations differ significantly, with Redfish generally offering higher accuracy (e.g. through the use of higher-bit formats, whereas IPMI often uses 8-bit raw numbers).

In the context of container power consumption estimation, IPMI implementations occupy an interesting role. In 2016, Kavanagh et al.[11] found the accuracy of IPMI power data to be relatively low when compared with an external power meter, mainly due to the large measurement window size of 120 to 180 seconds and the inaccurate assessment of idle power. They concluded that IPMI power data was still useful when a longer averaging window was used and the initial data points were discounted. In a later study, they suggested combining the measurements of IPMI

and Intel RAPL (which they found to underestimate power consumption) for a reasonable approximation of true measurements[12]. Kavanagh's findings have been cited in various studies, often to argue against using IPMI for power measurement. When used, it is sometimes chosen simply because it was the "simplest power metric to read"[66] in the context of entire data centers.

Redfish is a modern out-of-band management system, first released in 2015 explicitly to replace IPMI[67]. It uses a RESTful API and JSON data format, making queries easier to perform via code. In 2019, Wang et al.[4] directly compared IPMI and Redfish power data to readings from a high-accuracy power analyzer and found Redfish to be more accurate than IPMI, with a MAPE of 2.9%, while also finding a measurement latency of about 200ms. They also found measurements to be more accurate in higher power ranges, which they attributed to the improved latency.

In conclusion, BMC power data acquired over Redfish provides a simple and comparatively easy way to measure system power based on various physical system sensors. Its greatest strength lies in easy implementation and general availability. In the context of container energy consumption, BMC power data lacks the short sampling rates necessary to measure a highly dynamic container setup but can prove useful as a validation or cross-reference dataset for longer intervals exceeding 120 seconds. Unfortunately, the data quality of BMC power data depends on the actual system, and power models can be significantly improved by initial calibration with an external power measurement device[11].

2.4 In-Band Measurement Techniques

In-band measurement techniques refer to methods of power consumption monitoring that utilize built-in telemetry capabilities of system components to collect energy usage data directly from within the host system. Unlike external power meters or BMCs like IPMI, which operate independently of the main system, in-band techniques leverage on-die sensors and software interfaces to gather power metrics in real-time. These techniques provide fine-grained data with minimal additional hardware, making them well-suited for scalable environments like Kubernetes clusters. However, their accuracy and granularity are often dependent on the hardware's internal estimation algorithms, which may introduce uncertainties compared to direct measurement methods.

2.4.1 ACPI

The *Advanced Configuration and Power Interface (ACPI)* is a standardized interface that facilitates power management and hardware configuration by allowing the operating system to control hardware states such as processor sleep, throttling, and performance modes [10]. It plays a significant role in processor performance tuning by exposing C-states (idle), P-states (performance), and T-states (throttling), which the OS can leverage to adjust the processor's activity, frequency, and voltage.

Although ACPI defines these power states, their actual implementation is processor-specific, and the interface does not provide real-time telemetry. As such, ACPI does not expose instantaneous power consumption values. Any attempt to estimate power based on ACPI would require detailed knowledge of processor-specific behavior, including the mapping between frequency, voltage, and power information

that is not exposed through ACPI. As a result, limited research has been conducted on this topic.

In theory, one could attempt to use ACPI's _PSS (Performance Supported States) table, which lists available P-states along with nominal voltage, frequency, and optionally estimated maximum power dissipation, to perform rough CPU power estimation. This method would involve tracking CPU residency in each performance state and applying simple integration models to estimate total energy. However, due to the static nature of _PSS entries and the lack of temporal precision, such estimates would be inherently coarse-grained and typically inaccurate for modern processors with dynamic voltage and frequency scaling or turbo modes.

Consequently, ACPI is rarely used in contemporary power estimation contexts. Its primary role remains in system configuration and power state control rather than accurate energy quantification. In modern Intel processors, the Running Average Power Limit (RAPL) interface provides a more appropriate solution for in-band power measurement. This makes RAPL the preferred tool for energy-aware computing research and production environments alike.

2.4.2 Intel RAPL

Intel Running Average Power Level (RAPL) is a Power Monitoring Counter (PMC)-based feature introduced by Intel that provides a way to monitor and control the energy consumption of various components within their processor package[68]. An adaptation of RAPL for AMD processors uses largely the same mechanisms and the same interface[69], although it provides less information than Intel's RAPL, offering no DRAM energy consumption[22]. Unfortunately, RAPL lacks detailed low-level implementation documentation, and the exact methodology of the RAPL calculations remains unknown[24].

Intel RAPL has been used extensively in research to measure energy consumption[20] despite some objections regarding its accuracy, which will be discussed in § 2.4.2.2 and § 2.4.2.3. The general consensus is that RAPL is *good enough* for most scientific work in the field of server energy consumption and efficiency. As Raffin et al.[16] point out, it is mostly used *like a black box without deep knowledge of its behavior*, resulting in implementation mistakes. For this reason, § 2.4.2.1 presents an overview of the RAPL fundamentals.

2.4.2.1 RAPL Measurement Methods

This subsection provides an overview of how RAPL works and is used. It is based on the Intel Architectures Software Developer's Manual[15, Section 16.10] and the works of Raffin et al. [16] (2024) and Schöne et al. [21] (2024).

Running Average Power Limit (RAPL) is a power management interface in Intel CPUs. Apart from power limiting and thermal management, it also allows measuring the energy consumed by various components (or *domains*). These domains include individual CPU cores, integrated graphics (in non-server CPUs), and DRAM, as well as the *package*, referring to the whole CPU die. While it initially used models to estimate energy use[17], it now uses physical measurements. The processor is divided into different power domains or "planes", representing specific components,

as seen in figure 2.1. Notably, not all domains are present in all systems: both client-grade systems feature the *Package* and *PP0 core* domains, server-grade processors typically do not show the *PP1 uncore* domain (typically used for integrated GPUs), and client-grade processors do not show the *DRAM* domain. The *PSYS* domain for the "whole machine" is ill-defined and only exists on client-grade systems. In an experiment with recent Lenovo and Alienware laptops, Raffin et al. found that the *PSYS* domain reported the total consumption of the laptop, including display, dedicated GPU, and other domains. Regardless, this thesis focuses on the RAPL power domains available to server-grade processors.

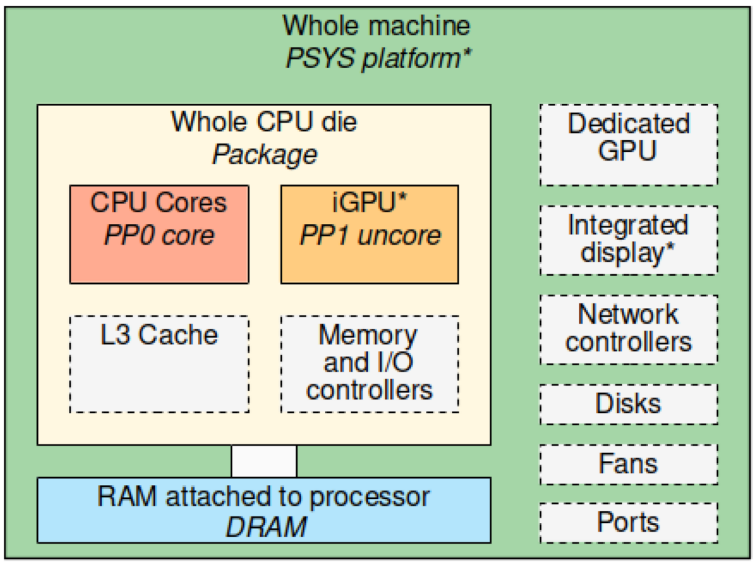


FIGURE 2.1: Hierarchy of possible RAPL domains and their corresponding hardware components. Domain names are in italic, and grayed items do not form a domain on their own. Items with an asterisk are not present on servers[16].

RAPL provides hardware counters to read the energy consumption (and set power limits) for each domain. The energy consumption is measured in terms of processor-specific "energy units" (e.g. 61μJ for Haswell and Skylake processors). The counters are exposed to the operating system through model-specific registers (MSRs) and are updated approximately every millisecond. The main advantages of RAPL are that no external power meters are required, nor privileged access to the BMC (which could be used to power off the server). RAPL is more accurate than any untuned statistical estimation model.

Various measurement methods can be used to extract RAPL measurements. In a detailed comparison, Raffin et al.[16] outline their individual features and tradeoffs, summarized in figure 2.2b:

- **Lacking documentation:** Since there is no publicly available documentation of the low-level RAPL implementation, implementations are bound to suffer inaccuracies and inconsistencies due to a lack of understanding.

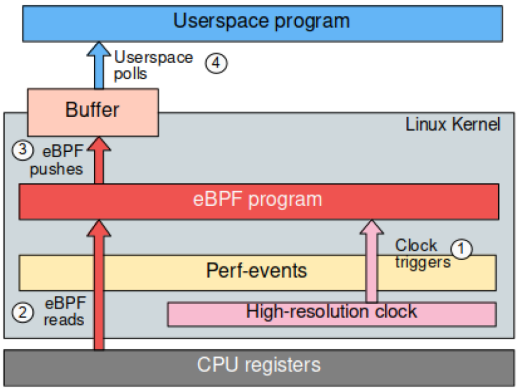
- The **Model-Specific Register (MSR)** interface provides low-level access to RAPL energy counters but is complex and hardware-dependent. Developers must manually determine register offsets and unit conversions based on processor model and vendor documentation. This method lacks safeguards, requires

deep processor knowledge, and is error-prone, with incorrect readings difficult to detect. Although read-only access poses no risk to system stability, MSRs expose sensitive data and are thus restricted to privileged users (e.g. `root` or `CAP_SYS_RAWIO`). Fine-grained access control is not supported natively, though the `msr-safe` module offers limited mitigation.

- The **Power Capping (powercap)** framework is a high-level Linux kernel interface that exposes RAPL energy data through the sysfs filesystem, making it accessible from userspace. It simplifies energy measurements by automatically handling unit conversions and domain discovery, requiring minimal hardware knowledge. Though domain hierarchy can be confusing (especially with DRAM domains appearing nested under the package domain), powercap remains user-friendly and scriptable. It supports fine-grained access control via file permissions and offers good adaptability to hardware changes, provided the measurement tool doesn't rely on hard-coded domain structures.

- The **perf-events** subsystem provides a higher-level Linux interface for accessing RAPL energy counters as counting events. It supports overflow correction and requires less hardware-specific knowledge than MSR. Each RAPL domain must be opened per CPU socket using `perf_event_open`, and values are polled from userspace. While it lacks a hierarchical structure like powercap and may be harder to use in certain languages or scripts, it remains adaptable and robust across different architectures. Fine-grained access control is possible via kernel capabilities or `perf_event_paranoid` settings.

- **eBPF** enables running custom programs in the Linux kernel, and in this context, it is used to directly read RAPL energy counters from within kernel space, potentially reducing measurement overhead by avoiding user-kernel context switches. The implementation attaches an eBPF program to a CPU clock event, using `perf_event_open` to access energy counters and buffering results for userspace polling (as visualized in figure 2.2a). While offering the same overflow protection as regular `perf-events`, this approach is significantly more complex, prone to low-level errors (especially in C), and requires elevated privileges (`CAP_BPF` or `root`). It also lacks portability, as it demands manual adaptation to kernel features and domain counts, limiting its maintainability across systems.



(A) RAPL perf-event eBPF mechanism

mechanism	technical difficulty	required knowledge	safeguards	privileges	resiliency
MSR	medium	CPU knowledge	none	SYS_RAWIO cap. + msr module	poor
perf-events + eBPF	high (long, complicated code)	limited	overflows unlikely, many other possible mistakes	PERFMON and BPF capabilities	manual tweaks necessary for adaptation
perf-events	low	limited	good, overflows unlikely	PERFMON capability	good
powercap	low	limited	beware of overflows	read access to one dir	good, very flexible

(B) RAPL measurement mechanisms comparison

FIGURE 2.2: RAPL measurements: eBPF and comparison[16]

In their research, Raffin et al. conclude that all four mechanisms have small or negligible impact on the running time of their benchmarks. They formulate the following recommendations for future energy monitoring implementations:

- Measuring frequencies should be adapted to the state of the node to prevent high measurement overhead caused by reduced time spent in low-power states. Under heavy load, a high frequency can be used to capture more information.
- `perf-events` is the overall recommended measurement method, providing good efficiency, latency, and overflow protection. Powercap is less efficient but provides a simpler sysfs API.
- Even though `perf-events` and the eBPF measurement method appear to be the most energy-efficient, they are not recommended due to their complexity. For the same reason, the MSR method is not recommended, as it raises complexity while counter-intuitively being slower than `perf-events`.

RAPL MSRs can be read on some cloud computing resources (e.g. some Amazon EC2 instances), although the hypervisor traps the MSR reads, which can add to the polling delay. In EC2, the performance overhead also significantly increases to <2.5% (compared to <1% on standalone systems)[24].

2.4.2.2 RAPL Validation

Since its inception, RAPL has been the subject of various validation studies, with the general consensus that its accuracy could be considered "good enough"[16]. Notable works include Hackenberg et al., who in 2013 found RAPL accurate but missing timestamps[18], and in 2015 noted a major improvement to RAPL accuracy after Intel switched from a modeling approach to actual measurements for their Haswell architecture[17]. Desrochers et al. concluded in a 2016 RAPL DRAM validation study[8] that DRAM power measurement was reasonably accurate, especially on server-grade CPUs. They also found measurement quality to drop when measuring an idling system. Later, Alt et al.[19] tested DRAM accuracy of heterogeneous memory systems of the more recent Ice Lake-SP architecture and concluded that DRAM estimates behaved differently than on older architectures. They noted that RAPL overestimates DRAM energy consumption by a constant offset, which they attribute to the off-DIMM voltage regulators of the memory system.

A critical point in RAPL validation was the introduction of the Alder Lake architecture, marking Intel's first heterogeneous processor, combining two different core architectures from the Core and Atom families (commonly referred to as P-Cores and E-Cores) to improve performance and energy efficiency. While this heterogeneity can improve performance and energy efficiency, it also increases the complexity of scheduling decisions and power-saving mechanisms, adding to the already complex architecture, which features per-core Dynamic Voltage and Frequency Scaling (DVFS), idle states, and power limiting/thermal protection.

Schöne et al.[21] found RAPL in the Alder Lake architecture to be generally consistent with external measurements but exhibiting lower accuracy in low-power scenarios. The following figure 2.3 shows these inaccuracies, albeit tested on a consumer-grade Intel Core i9-12900K processor measured at the base frequency of 0.8GHz.

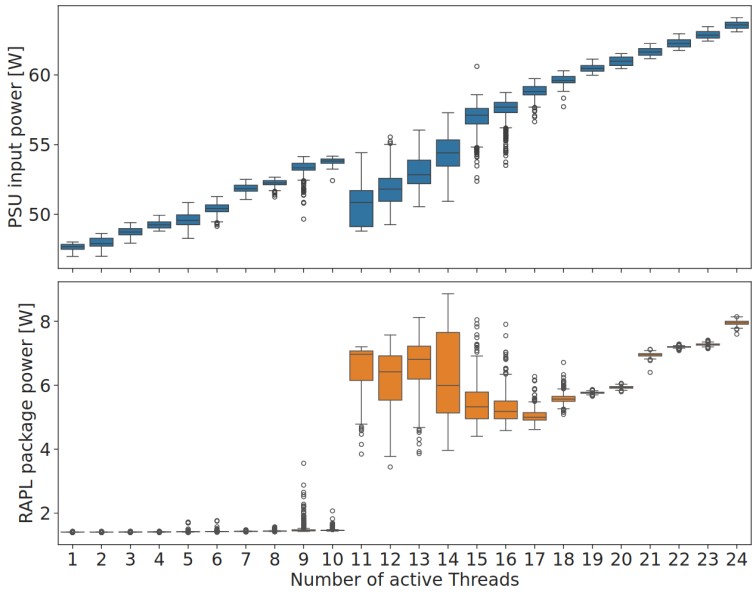


FIGURE 2.3: RAPL and reference power consumption sampled at 100 ms / 50 ms intervals, respectively. Double precision matrix multiplication kernel at 0.8GHz running for 60s each at increasing number of active threads[21].

2.4.2.3 RAPL Limitations and Issues

Several limitations of RAPL have been identified in various research works. Since RAPL is continually improved by Intel as new processors are released, some of these issues have since been improved or entirely solved.

- **Register overflow:** The 32-bit register can experience an overflow error[16, 23]. This can be mitigated by sampling more frequently than the register takes to overflow. This interval can be calculated using the following equation:

$$t_{\text{overflow}} = \frac{2^{32} \cdot E_u}{P} \tag{2.1}$$

  Here, E_u is the energy unit used (61μJ for Haswell), and P is the power consumption. On a Haswell processor consuming 84W, an overflow would occur every 52 minutes. Intel acknowledges this in the official documentation, stating that the register has a *wraparound time of around 60 seconds when power consumption is high*[15]. This is solvable with a simple correction, provided that the measurement intervals are small enough: for two successive measurements m_{prev} and m_{current}, the actual measured difference is given by

$$\Delta m = \begin{cases} m_{\text{current}} - m_{\text{prev}} + C & \text{if } m_{\text{current}} < m_{\text{prev}} \\ m_{\text{current}} - m_{\text{prev}} & \text{otherwise} \end{cases} \tag{2.2}$$

  where C is a correction constant that depends on the chosen mechanism:

- **DRAM accuracy:** DRAM accuracy can only reliably be used for the Haswell architecture[8, 19, 23] or newer, and may still exhibit a constant power offset (attributed to the voltage regulator power loss of the memory system).

Mechanism	Constant C
MSR	`u32::MAX`, i.e., $2^{32} - 1$
perf-events	`u64::MAX`, i.e., $2^{64} - 1$
perf-events with eBPF	`u64::MAX`, i.e., $2^{64} - 1$
powercap	Value given by the file `max_energy_uj` in the sysfs folder for the RAPL domain

TABLE 2.2: RAPL overflow correction constant

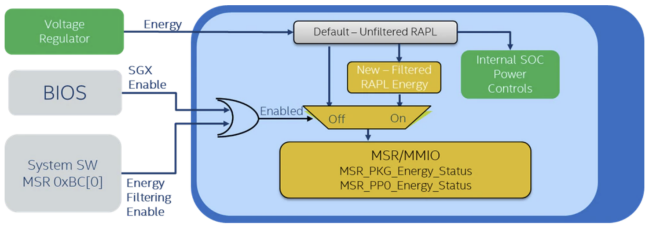
- **Unpredictable timings:** While Intel documentation states that the RAPL time unit is 0.976ms, the actual intervals may vary. This is an issue since the measurements do not include timestamps, making precise measurements difficult[23]. Several coping mechanisms have been used to mitigate this, notably *busypolling* (polling the counter for updates, significantly compromising overhead in terms of time and energy[70]), *supersampling* (lowering the sampling interval, increasing overhead and occasionally creating duplicates that need to be filtered[23]), or *high-frequency sampling* (lowering the sampling rate when the resulting data is still sufficient[71]). Another solution is to use a *low sampling frequency* to smooth out the relative error due to spikes, with the only drawback of a loss of temporal precision. At sampling rates slower than 50Hz, the relative error is less than 0.5%[24].

- **Non-atomic register updates:** RAPL register updates are non-atomic[23], meaning that the different RAPL values show a delay between individual updates. This may introduce errors when sampling multiple counters at a high sampling rate, making it possible to read both fresh and stale values of different counters.

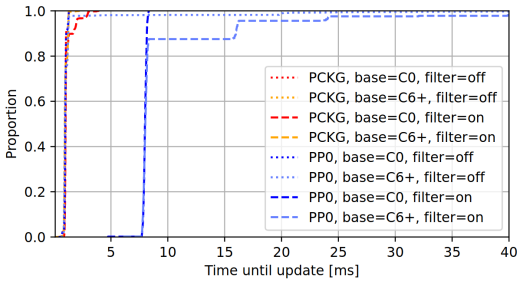
- **Lower idle power accuracy:** When measuring an idling server, RAPL tends to be less accurate[8, 21].

- **Side-channel attacks:** While the update rate of RAPL is usually 1ms, it can get as low as 50μs for the PP0 domain (processor cores) on desktop processors[21]. This can be exploited to retrieve processed data in a side-channel attack (coined "Platypus")[21, 25].

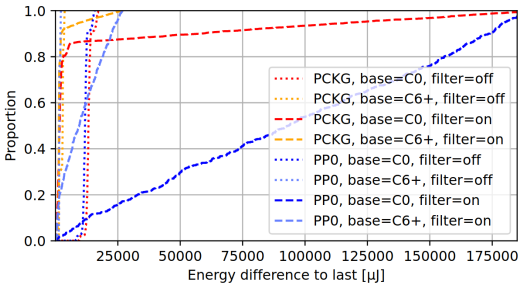
  To mitigate this issue while retaining RAPL functionality, Intel implements a filtering technique via the `ENERGY_FILTERING_ENABLE`[26, Table 2-2] entry or when *Software Guard Extension (SGX)* is activated in the BIOS. This filter adds random noise to the reported values (visualized in figure 2.4a). For the PP0 domain, this raises the temporal granularity to about 8ms. While this does not affect the average power consumption, point measurement power consumption can be affected. Figure 2.4 shows the effect of the filter, clearly indicating the loss of granularity resulting from its activation. In a 2022 article, Tamara[72] found a surprisingly higher mean with the filter activated and deemed filtered RAPL energy data unusable. In a more elaborate experiment in 2024, Schöne et al. did not encounter these inaccuracies anymore.



(A) RAPL energy filtering[73]



(B) Distribution of time between updates: With an enabled filter, the PP0 domain only provides updates every 8 ms; otherwise, RAPL values are updated every 1 ms. If the load is too low, some updates might be skipped, e.g. the next update for PP0 with an enabled filter is at 16 ms.



(C) The minimal increase of a measurement is 1 energy unit of 61.035 μJ. Enabling the filter significantly influences measurements for the PP0 domain and measurably affects PCKG measurements.

FIGURE 2.4: Observable loss of granularity caused by the activation of `ENERGY_FILTERING_ENABLE`[21]

2.4.2.4 RAPL Conclusions

The energy measurement accuracy of RAPL has significantly improved since its inception and provides a generally accepted way to measure system energy consumption. It is well-validated and accepted as the most accurate fine-granular energy measurement tool. Some known limitations have historically caused inaccuracies in developed measurement tools, but corrections for these limitations exist.

2.4.3 Graphical Processing Units (GPU)

In recent years, the utilization of GPUs in cloud computing environments has grown significantly, driven primarily by the increasing demand for high-performance computations in machine learning, artificial intelligence, and large-scale data processing[74]. Kubernetes now includes mechanisms for GPU provisioning, enabling containerized workloads to leverage GPU acceleration[75].

Although GPUs remain less common than traditional CPU-based workloads in typical Kubernetes clusters, their adoption is rapidly accelerating. Industry reports indicate that GPU usage in Kubernetes has seen a growth rate of nearly 58% year-over-year, outpacing general cloud computing growth rates[76]. This increase is largely attributed to ML workloads and real-time processing tasks that benefit from the parallel processing capabilities of GPUs[77]. Furthermore, hyperscalers have integrated GPU support directly into their managed Kubernetes services, reflecting the growing demand for GPU-powered workloads in containerized environments.

Despite this growth, GPU deployments are still not as pervasive as CPU-based workloads in Kubernetes-managed clusters. The primary focus of this thesis is on the

measurement and analysis of energy consumption in more common CPU- and memory-centric Kubernetes workloads. Nevertheless, due to the rising significance of GPUs, their energy measurement techniques and potential integration within Kubernetes environments are briefly examined.

Ultimately, the inclusion of GPU energy measurements remains outside the primary scope of this thesis but is acknowledged as an important area for future research. This structured exploration serves to highlight current limitations and opportunities for enhancing energy efficiency in Kubernetes-managed GPU workloads.

2.4.3.1 GPU Virtualization Technologies

Full GPU Virtualization Full GPU virtualization provides isolated instances of a single physical GPU to multiple virtual machines. This is achieved using technologies such as NVIDIA's *vGPU* or AMD's *MxGPU (Multiuser GPU)*. These technologies allow a VM to see a complete GPU, while the underlying hypervisor manages resource partitioning and scheduling[78, 79], either through partitioning or time-slicing. In a Kubernetes environment, full GPU virtualization is commonly utilized through:

- **vGPU on VMware or OpenStack:** Kubernetes clusters running on VMware vSphere or OpenStack can request vGPU instances as if they were physical GPUs. These instances are shared among containers while maintaining memory and compute isolation.
- **Device Plugin Integration:** NVIDIA, AMD, and Intel provide a Device Plugin for Kubernetes, enabling seamless GPU discovery and allocation across pods[75].

Multi-Instance GPU (MIG) Introduced with the NVIDIA A100 architecture, Multi-Instance GPU (MIG) allows a single GPU to be partitioned into up to seven independent instances, each with its own dedicated compute, memory, and cache resources[80]. Unlike traditional vGPU, MIG provides true hardware-level isolation, preventing noisy-neighbor effects and enabling finer resource allocation. MIG instances are exposed to Kubernetes as individual GPUs. For example, a single A100 GPU partitioned into seven MIG instances appears as seven separate GPU resources, each assignable to different containers. MIG-aware device plugins ensure proper scheduling and isolation. Hence, MIG technology is particularly useful for multi-tenant environments and supports finer granularity in resource allocation compared to traditional vGPU models.

GPU Passthrough GPU passthrough allows a physical GPU to be exclusively assigned to a single VM or container. Unlike virtualization, where resources are shared, passthrough dedicates the full GPU to one environment, offering near-native performance[81]. GPU passthrough is configured at the hypervisor level (e.g. KVM or VMware ESXi) and can be exposed to Kubernetes nodes. Pods scheduled on nodes with GPU passthrough access gain complete control of the GPU, enabling direct memory access and high-performance computation.

GPU virtualization technologies enable efficient multi-tenant use of GPU resources, enhancing performance and cost-effectiveness in cloud-native environments. For

the purposes of energy measurement, understanding these virtualization layers is essential for accurate per-container energy attribution.

2.4.3.2 GPU NVIDIA-NVML Energy Measurements and Validation

Modern GPUs are equipped with **built-in power sensors** that enable real-time energy measurement. For instance, NVIDIA GPUs expose power metrics through the *Nvidia System Management Interface (nvidia-smi)*, which reports instantaneous power draw, temperature, and memory usage[82]. This interface allows for programmatic access to GPU power consumption, making it a common choice for monitoring and energy profiling in both standalone and containerized environments[80].

In 2024, Yang et al. conducted a comprehensive study on the accuracy and reliability of NVIDIA's built-in power sensors, examining over 70 different models[27]. They concluded that previous research placed excessive trust in NVIDIA-NVML, overlooking the importance of measurement methodology. The study revealed several critical findings:

- **Sampling Limitations:** NVIDIA NVML offers the option to specify a sampling frequency in units of milliseconds. However, on certain models, such as the A100 and H100, power is sampled only around 25% of the time, introducing potential inaccuracies in total energy consumption estimations.
- **Transient Response Issues:** While measured power reacted instantly to a suddenly applied workload, NVIDIA-NVML would report values with a delay of several hundred milliseconds on some devices. Additionally, a slower rise (with linear growth) was observed, taking over a second to reach correct power figures in some instances. Generally, server-grade GPUs were shown to provide more instantaneous power measurements.
- **Measurement Inaccuracies:** The average error rate in reported power draw was found to be approximately 5%, deviating from NVIDIA's claimed fixed error margin of 5W. This error remained consistent when the GPU reached a constant power draw.
- **Averaging Effects:** Reported power consumption values are averaged over time, masking short-term fluctuations and potentially underreporting peak consumption.

To address these limitations, the study proposed best practices such as running multiple or longer iterations of workloads to average out sampling errors, introducing controlled phase shifts to capture different execution states, and applying data corrections to account for transient lags[27]. These adjustments reduced measurement errors by up to 65%, demonstrating the importance of refining raw sensor data for more accurate energy profiling.

2.4.3.3 Related Research

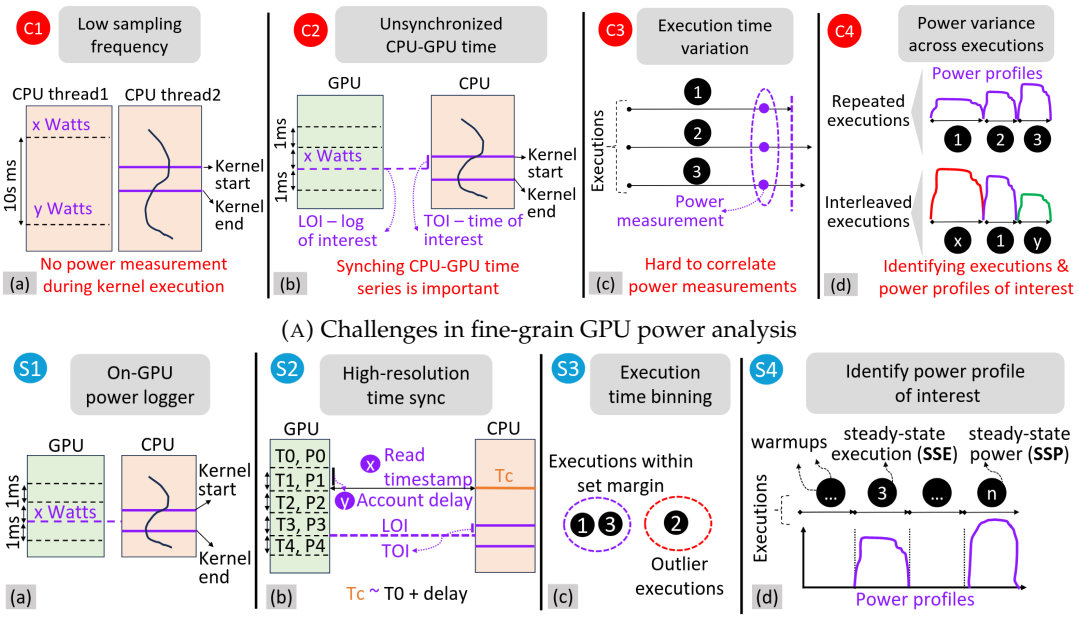
While most research has used NVIDIA-NVML to measure GPU power consumption, some research has focused on alternative measurement tools, usually to address issues similar to those stated by Yang et al. in the previous section. Specifically,

the following three tools were proposed to provide higher sampling rates to enable finer-grained power analysis.

AccelWattch In 2021, Pan et al. proposed *AccelWattch*[83], a configurable GPU power model that provides both a higher accuracy cycle-level power model and a way to measure constant and static power, utilizing any pure-software performance model, NVIDIA-NVML, or a combination of the two. Notably, their model is DVFS-, power-gating-, and divergence-aware. The resulting power model was validated against measured ground truth using an NVIDIA Volta GV100, yielding a MAPE error between $7.5 - 9.2 \pm 2.1\% - 3.1\%$, depending on the AccelWattch variant. The Volta model was later validated against Pascal TITAN X and Turing RTX 2060 architectures without retraining, achieving $11 \pm 3.8\%$ and $13 \pm 4.7\%$ MAPE, respectively. The authors conclude that AccelWattch can reliably predict power consumption of these specific GPU architectures. In the context of Kubernetes energy consumption, AccelWattch contributes a fine-grained temporal granularity.

FinGraV In 2024, Singhania et al. proposed *FinGraV*[84] (abbreviated from **Fin**e-**Gra**in **V**isibility), a fine-grained power measurement tool capable of sub-millisecond power profiling for GPU executions on an AMD MI300X GPU. They identified the following main challenges of high-resolution GPU power analysis (see figure 2.5a):

- **Low Sampling Frequency:** Standard GPU power loggers operate at intervals too coarse (tens of milliseconds) to capture the sub-millisecond executions of modern kernels.
- **CPU-GPU Time Synchronization:** Synchronizing power measurements with kernel start and end times is problematic due to the asynchronous nature of CPU-GPU communication.
- **Execution Time Variation:** Minor variations in memory allocation or access patterns lead to inconsistent kernel execution times, complicating time-based power profiling.
- **Power Variance Across Executions:** Repeated executions of the same kernel, or interleaved executions with other kernels, result in fluctuating power consumption, challenging consistent profiling.



(A) Challenges in fine-grain GPU power analysis

(B) FinGraV strategies to address challenges

FIGURE 2.5: FinGraV GPU power measurement challenges and strategies[84]

To overcome these challenges, FinGraV introduces several strategies (see figure 2.5b):

- **On-GPU Power Logger:** FinGraV leverages a high-resolution (1 ms) power logger, capturing the average of multiple instantaneous power readings.
- **High-Resolution Time Synchronization:** GPU timestamps are read from the CPU side before kernel execution, and synchronization is maintained throughout execution to correlate power samples with kernel events.
- **Execution Time Binning:** Kernel executions are grouped into "bins" based on empirical runtime ranges, enabling tighter power profiling while discarding outlier runs.
- **Power Profile Differentiation:** FinGraV distinguishes between Steady-State Execution (SSE) and Steady-State Power (SSP) profiles. SSP represents the stabilized power consumption after initial transients, providing the most accurate depiction of kernel power consumption.

The application of FinGraV to benchmarks reveals several critical observations: Kernel executions differ significantly between initial runs and steady-state, with deviations of up to 80%. Memory-bound kernels and compute-light kernels are found to be highly sensitive to the preceding kernel, impacting their power profile. Furthermore, the authors expose discrepancies in GPU power scaling relative to computational load, particularly for compute-light kernels.

FinGraV introduces promising concepts that could, in theory, enable more granular and accurate GPU power analysis in container-based GPU workloads. Its methodological approach addresses key challenges in sub-millisecond power measurement. However, its current implementation is tightly coupled with the AMD MI300X GPU,

relying on hardware-specific logging capabilities that are not universally available. While the underlying concepts may be extendable to other GPUs, achieving this is far from trivial, requiring significant adaptation and low-level access to power metrics that are often proprietary or limited by driver capabilities.

Consequently, FinGraV highlights both the challenges and potential solutions for fine-grained GPU power analysis but falls short of providing a general-purpose framework that could be easily integrated into Kubernetes energy measurement tools. It also underscores the broader issue that GPU energy consumption analysis remains relatively immature, with only vendor-specific tools like NVIDIA-NVML offering practical (but coarse) power metrics. This illustrates that while the methodology is theoretically sound, practical implementation across diverse GPU architectures remains a significant challenge.

PowerSensor3 *PowerSensor3*[9] is an open-source hardware tool introduced in 2025, designed to provide high-resolution power measurements for GPUs, SoC boards, PCIe devices, SSDs, and FPGAs. Unlike software-based power models or vendor-specific tools such as NVIDIA's NVML, PowerSensor3 achieves significantly higher accuracy and granularity through direct voltage and current measurements at a sampling rate of up to 20 kHz. This fine temporal resolution allows it to capture transient power behaviors that are typically missed by software-based methods, which are constrained by lower sampling frequencies and indirect estimations. As expected for a purpose-built hardware solution, PowerSensor3 outperforms NVML in both precision and the ability to detect rapid changes in power consumption.

A particularly valuable feature of PowerSensor3 is its capability to monitor not only GPUs but also other critical components such as SoC boards, PCIe-connected accelerators, and storage devices like SSDs. For Kubernetes-based energy efficiency analysis, this would provide unprecedented visibility into the power usage of individual containers, extending monitoring beyond the CPU and GPU to the broader spectrum of peripherals that contribute to overall energy consumption. Such granularity could enhance resource scheduling and energy optimization in containerized environments.

However, while its technical benefits are evident, the practical deployment of dedicated hardware sensors like PowerSensor3 at scale remains both complex and expensive. Integrating such devices across large Kubernetes clusters would require substantial investment in hardware and reconfiguration of infrastructure, making wide adoption unlikely outside of specialized research environments. Consequently, PowerSensor3 and other hardware-dependent methods are not considered in the scope of this thesis. Furthermore, the very recent introduction of PowerSensor3 in 2025 highlights the ongoing challenges of accurate energy monitoring through software alone, reflecting the current gap in reliable, scalable, software-based power measurement solutions.

2.4.3.4 GPU Limitations in Kubernetes Context

The analysis of GPU power consumption has revealed promising research efforts aimed at achieving fine-grained power visibility and energy optimization. Tools such as FinGraV and PowerSensor3 demonstrate that significant strides are being

made in capturing detailed power metrics with high temporal resolution and sub-component granularity. FinGraV addresses the complexities of short-lived GPU kernel executions through innovative profiling methodologies, while PowerSensor3 delivers hardware-level accuracy for GPUs, SoC boards, and various PCIe-connected peripherals. These solutions underscore the potential for more refined power monitoring in high-performance GPU workloads.

However, the current state of GPU energy consumption measurement presents significant challenges for scalable, container-based energy tracking in Kubernetes environments. Research tools like FinGraV and PowerSensor3, while technically robust, are either hardware-dependent or too tightly coupled to specific architectures (such as AMD's MI300X in the case of FinGraV). Hardware-based solutions like PowerSensor3, though highly accurate, are impractical for widespread deployment due to cost and scalability concerns. Meanwhile, software-based vendor solutions such as NVIDIA's NVML are far more accessible but suffer from limitations in temporal granularity and measurement accuracy. These tools offer convenient integration and broad support across data center infrastructures but struggle with capturing rapid transients in power consumption, which are crucial for real-time container energy attribution.

In the context of this thesis, GPU energy consumption is acknowledged as an important yet currently impractical aspect of container energy measurement. The relative immaturity of fine-grained, scalable monitoring solutions for GPUs, combined with the relatively small role of GPUs in Kubernetes clusters, justifies this exclusion. Although the utilization of GPU accelerators in Kubernetes environments is expected to grow, current measurement methods do not yet support the level of precision and scalability required for effective implementation. As such, this thesis will focus on more readily measurable server components, with the understanding that future advancements in GPU power analysis may enable their integration into Kubernetes-based energy efficiency strategies.

2.4.4 Storage Devices

Various studies have investigated the power consumption of storage devices. In 2008, Hylick et al.[85] investigated real-time HDD energy consumption and found significant differences in power consumption between standby, idle, and active power states. Cho et al.[38] proposed various energy estimation models for SSDs after measuring and comparing the energy consumption of different models. The most notable model-based energy consumption estimation mechanisms are presented in § 2.5.4.

In contrast to CPU or GPU components, storage devices (HDD, SSD, or NVMe drives) cannot make use of physical power sensors. While a BMC-measurement-based solution would technically be feasible, real-world implementation is impractical: while a BMC might be able to measure the power supply to a storage device, it typically is not exposed through IPMI or Redfish. Such measurements would further be complicated by the use of backplane devices, making measurements for individual devices impossible. For these reasons, storage device energy consumption is typically modelled, not measured (see § 2.5.4).

While storage devices don't expose any energy-consumption-specific metrics, many other related metrics are available (and can be used for modeling approaches):

- `NVMe-cli`[37] exposes many metrics of NVMe drives, including the maximum power draw for each power state (including idle power), the number of power states supported, the current power state and temperature, and others.
- `smartctl`[36] exposes metrics of the *SMART (Self-Monitoring, Analysis and Reporting Technology)* functionality implemented in many modern storage drives. While these metrics are vendor-specific, they often include temperature, throughput, and other indicators. Often, HDD speed is exposed. Notably, *SMART* metrics are typically more focused on lifecycle information such as power-on hours, wear indicators, and others.
- Many other performance metrics are exposed by various tools such as `iostat`, `sar`, `/proc/diskstats`, and `blkstat`, such as read/write IOPS, throughput, queue length, latency, utilization, and others. Additional information (such as the interface) is also exposed.

2.4.5 Network Devices and Other PCIe Devices

Peripherals like the Network Interface Card (NIC) are almost always connected via PCIe. As such, many cards support device power states[44] as specified by the PCIe specifications. Notably, not all NICs support all (or any) power states. These device states allow the server and device to negotiate a power state for the device, which typically means choosing a trade-off between power consumption and wake-up latency. For devices, PCIe specifies the following device states:

- D0 state (Fully on)
- D1 and D2 states (Intermediate power states)
- D3 state (Off state), with the distinction between D3hot and D3cold

Unfortunately, device power states are not in any way related to physical power specifications: while a specific power state might be useful for simple deductions (e.g. if a device is idling or active), no power figures can be deduced. In the event that a device's idle or maximum power is known, power states might potentially be used for a first estimation (i.e., an idling device is unlikely to consume its specified maximum power, and vice versa), but since a device's power characteristics cannot be reliably estimated (especially beyond just NICs), device power states cannot be used to reliably estimate device power consumption. An attempt at the estimation of NIC power consumption is covered in § 2.5.5.

2.5 Model-Based Estimation Techniques

In the absence of actual power data, power consumption models can be formulated that map variables (such as CPU or memory utilization) related to a server's state to its power consumption.

Due to the strong correlation between CPU utilization and server power, a great number of models use CPU metrics as the only indicator of server power. Fan et al.[45] proposed a linear interpolation between idle power and full power, which

they further refined into a non-linear form, with a parameter γ to be fitted to minimize mean square error. Similar research was done to further reduce error by introducing more complex non-linear models, such as Hsu and Poole[46], who studied the SPECpower_ssj2008 dataset of systems released between December 2007 and August 2010, and suggested the adaptation of two non-linear terms:

$$P_{\text{server}} = \alpha_0 + \alpha_1 u_{\text{cpu}} + \alpha_2 \left(u_{\text{cpu}}\right)^{\gamma_0} + \alpha_3 \left(1 - u_{\text{cpu}}\right)^{\gamma_1} \tag{2.3}$$

The division of server power consumption into idle (generally static) and dynamic power (modeled with many different methods throughout related research) has historically been a popular suggestion[86]. Other broadly similar attempts to model server energy consumption based on only a few variables exist, such as modeling server consumption based on CPU frequency[87].

2.5.1 Component-Level Power Models

While models like the ones listed above might work well when custom-fitted to specific, multi-purpose servers, they have since been surpassed by the more common approach of modelling server power as an assembly of its components, as Song et al.[47] propose:

$$P_{\text{server}} = P_{\text{cpu}} + P_{\text{memory}} + P_{\text{disk}} + P_{\text{NIC}} + C \tag{2.4}$$

where C denotes the server's base power, which includes the power consumption of other components (regarded as static). This approach can easily be extended to include various other components such as GPUs, FPGAs, or other connected devices.

2.5.1.1 Advantages and Disadvantages of Component-Level Power Models

A component-based approach to modeling server power consumption offers increased granularity and adaptability across a diverse range of server architectures. Modern data centers deploy heterogeneous hardware configurations optimized for specific workloads, such as CPU-intensive computing nodes, GPU-accelerated servers for machine learning, or memory-rich systems for in-memory databases. These configurations lead to vastly different power distribution profiles across components[88]. By modeling the energy consumption of individual components, it becomes possible to reflect these structural differences more accurately. Additionally, such models can reveal energy characteristics that would be obscured in aggregate metrics (for instance, a workload that imposes significant stress on storage devices without engaging the CPU may go undetected in simplistic, CPU-centric models). Finally, component-level analysis enables more precise evaluation of energy optimization techniques: the impact of mechanisms like dynamic voltage and frequency scaling (DVFS) or idle power states can be assessed not just in isolation but in terms of their contribution to overall server efficiency.

Despite offering finer granularity, component-based power modeling faces several inherent challenges. While servers are composed of individual components, they function as tightly integrated systems in which no component operates in isolation. The power consumption of one subsystem often depends on the behavior of others (for example, memory access patterns can influence CPU power states,

and I/O activity may trigger CPU wake-ups or increased cache usage). These inter-component interactions are difficult to capture accurately and are frequently overlooked in component-level models[2], leading to potentially misleading or incomplete estimations. Furthermore, the development of detailed and accurate models for each component is significantly more complex than holistic server-level modeling. Such models often require extensive empirical data, sophisticated estimation techniques, and continuous updates to remain valid across hardware generations. This not only increases the research burden but also demands a higher level of expertise for interpretation and practical application compared to simpler, utilization-based models.

2.5.2 CPU

Existing CPU power models generally model CPU power as a combination of other, existing power figures. Fan et al.[45] propose a linear interpolation between idle power and full power based on CPU utilization. Basmadjian et al.[89] observe that individual cores in a multi-core CPU can be modeled as individual cores, in addition to an overall CPU idle consumption. Non-linear models are also widely adopted, with Lou et al.[90] proposing a polynomial model as a univariate function of CPU utilization. Other models include individual CPU components[91, 92] for more fidelity.

While these models may be helpful to examine the dynamics of CPU power consumption in relation to different inputs, they are not helpful in finding CPU power consumption of an unknown CPU, i.e., without previously known idle or maximum power consumption. The existence of a model capable of accurately estimating CPU power consumption based solely on generalizable input factors (such as utilization or frequency) is questionable due to the great variance in architectures and technologies, as well as technological progress. Unsurprisingly, the author of this thesis was not able to find a model to estimate CPU power consumption.

2.5.3 Memory

Many memory power models have been proposed in the literature, many of them with an idle and a dynamic component of memory power consumption. While the idle power consumption is generally assumed to be known, dynamic memory power consumption has been modeled to depend on memory usage[93] or memory accesses[94], the number of cache misses[95], memory state[89], or other factors. Similar to the CPU models presented in the previous section, these models do not propose generalizable models able to predict memory consumption in situations where idle or maximum memory consumption is not previously known, instead focusing on examining power consumption dynamics. The same objections preventing generalizable CPU models apply to memory models as well, namely great variety, specialization, and technological progress. As a result, no model-based approaches exist that are capable of accurately estimating memory power consumption based solely on performance metrics.

2.5.4 Storage Devices

2.5.4.1 Generalization-Based Estimation

There is an urgent need in the storage industry for research into the area of workload-dependent power estimation[96]. Estimating the energy consumption of a storage device is challenging, especially due to the great variation between different devices. Some of these model variables can be determined on a running server system (e.g. device type, I/O operation type, access pattern, workload intensity, state, and more), while other variables are unknown to the server (e.g. Flash Transition Layer and flash factor, NAND organization, garbage collection, and more). A large storage device market has led to a high variation in devices with sometimes drastically different target uses (e.g. low-latency storage devices, high-concurrency storage devices, low-power storage devices).

Storage controllers further complicate the energy consumption estimation of storage devices by introducing an additional layer of abstraction between the operating system and the physical storage hardware. Their internal operations consume energy independently of the actual read/write workload observed by the host system. This makes it difficult to directly correlate application-level I/O activity with actual device-level power usage. Moreover, in many server configurations, multiple drives are managed behind a single controller, obscuring per-device energy attribution and introducing variability that model-based estimations often cannot accurately capture.

Scope Clarification In this thesis, only storage devices physically installed in the server are considered for power estimation. This includes devices such as HDDs, SSDs, and NVMe drives directly attached to the server. Dedicated external storage systems such as Storage Area Networks (SAN) or Network-Attached Storage (NAS) are not within the scope of this analysis. While such systems are important in data center environments, their energy consumption is not attributable at the granularity required for the workload-level estimation pursued in this thesis.

As a result, research into storage device energy consumption measurement that is generally applicable to all devices has been limited. For practical applications, generalizations are often used, such as the following tables 2.3a to 2.3d. While these approximations cannot be used in the context of this thesis, they may serve as an initial guideline.

HDD Type	Read/Write Power (W)	Idle Power (W)	Standby Power (W)
HDD (2.5″ SATA)	1.5 – 3.0	0.5 – 1.2	0.1 – 0.3
HDD (3.5″ SATA)	6 – 12	4 – 8	0.5 – 2.0
HDD (Enterprise)	7 – 15	5 – 10	0.5 – 2.5

(A) Typical HDD power consumption[97]

HDD Type	Read/Write Power (W)	Idle Power (W)	Standby Power (W)
5400 RPM HDD	6 – 9	4 – 6	0.5 – 1.5
7200 RPM HDD	8 – 12	6 – 8	0.6 – 1.8
10,000+ RPM HDD	10 – 16	8 – 12	1.0 – 2.5

(B) Common HDD RPM power consumption[97]

SSD Type	Read Power (W)	Write Power (W)	Idle Power (W)
2.5″ SATA	4.5 – 8	4.5 – 8	0.30 – 2
mSATA	1 – 5	4 – 8	0.20 – 2
M.2 SATA	2.5 – 6	4 – 9	0.40 – 2

(C) Typical SATA SSD power consumption[98]

NVMe Type	Read/Write Power (W)	Peak Power (W)	Standby Power (W)
M.2 NVMe PCIe 3.0	3 – 5	6 – 9	0.4 – 1.5
M.2 NVMe PCIe 4.0	5 – 7	8 – 12	0.5 – 2
M.2 NVMe PCIe 5.0	8 – 12	12 – 18	0.8 – 3

(D) Typical NVMe SSD power consumption[98]

TABLE 2.3: Power consumption for various storage device types.

Apart from simple estimations like those shown in tables 2.3a to 2.3d, a few works have focused on the energy consumption of individual storage devices. In 2015, Cho et al. developed Energysim[38], an SSD energy modeling framework advancing the understanding of component-level (i.e., the subcomponents of a storage device) energy consumption in storage devices. Its validation against real-world SSD measurements using an Intel X25-M yielded a less than 8% error. The work underscores the difficulty of modeling storage energy accurately due to high variability across architectures and workloads. Unfortunately, Energysim uses many model parameters such as NAND organization, idle and active current consumption, and as a result cannot be generalized to other storage devices where these are unknown.

In 2014, Li and Long[39] presented a workload-aware modeling framework to estimate the energy consumption of storage systems, challenging the assumption that SSDs are inherently more energy-efficient than HDDs. By classifying I/O workloads into capability workloads (performance-driven) and capacity workloads (storage size-driven), they developed mathematical models that account for the number of devices needed, workload execution time, and device power states (active, idle, standby). Their validation, based on empirical measurements using Seagate HDDs and a Samsung SSD, shows that SSDs are generally more efficient for high-performance workloads, while HDDs can outperform SSDs in archival or low-access

scenarios, particularly when effective power management (e.g. spin-down) is employed. Unfortunately, similar to the research by Cho et al., the presented models make use of various non-generalizable variables, most notably a device's idle, standby, and busy power consumption. In the context of this thesis, these are unknown, and the presented model consequently cannot be applied.

2.5.4.2 GSPN Modeling for Hybrid Storage Systems (Active Power States)

In 2022, Borba et al.[40] proposed a number of models based on generalized stochastic Petri nets (GSPN) for performance and energy consumption evaluation of individual and hybrid (HDD + SSD) storage systems. GSPN is a suitable formalism for storage system design as, unlike queueing network models, synchronization, resource sharing, and conflicts are naturally represented. Also, the phase approximation technique may be applied for modeling non-exponential activities, and events with zero delays (e.g. workload selection) may adopt immediate transitions.

The authors propose a single-storage model (either for a single storage device or a hybrid system as a blackbox) and a multiple-storage model.

The hybrid storage power consumption model proposed by Borba is parameterized by I/O type (read/write), access pattern (sequential/random), object size (4KB, 1MB), and thread concurrency. The model explicitly incorporates power consumption per operation (e.g. random-read-4KB on SSD).

The following notation is adopted:

- $E\{\#p\}$ represents the mean value of the inner expression, in which $\#p$ denotes the number of tokens in place.
- $W(T)$ represents the firing rate associated with transition T.
- $\eta : T_{\text{imm}} \rightarrow [0,1]$ maps each immediate transition ($t \in T_{\text{imm}}$) to a normalized weight. Weights represent the transition firing probability in a conflict set.
- $pRequests(N)$ denotes the amount of concurrent requests from simultaneous clients (workers).

Single-device storage energy consumption is estimated as follows:

$$\begin{aligned} EP_w = \kappa \cdot (&EP_{w1} \cdot \alpha \cdot \beta + EP_{w2} \cdot (1-\alpha) \cdot \beta \\ &+ EP_{w3} \cdot \alpha \cdot (1-\beta) + EP_{w4} \cdot (1-\alpha) \cdot (1-\beta)) \end{aligned} \tag{2.5}$$

$$\begin{aligned} EP_r = (1-\kappa) \cdot (&EP_{r5} \cdot \alpha \cdot \beta + EP_{r6} \cdot (1-\alpha) \cdot \beta \\ &+ EP_{r7} \cdot \alpha \cdot (1-\beta) + EP_{r8} \cdot (1-\alpha) \cdot (1-\beta)) \end{aligned} \tag{2.6}$$

$$EC = (EP_w + EP_r) \cdot TH \cdot \text{time} \tag{2.7}$$

where EP_w and EP_r are the mean power consumption for a read (r) or write (w) operation, which is estimated using the mean power of each workload feature. For instance, EP_{w1} denotes the power of a write operation (w) using random access (α) and a small object (β). System throughput (i.e., IOPS) is estimated as $TH = E\{\#p_{\text{Ack}}\} \times W(t_{\text{Communicating}})$. For the single-device model, the following weights are taken into

account: $\eta(t_{\text{Write}}) = \kappa$; $\eta(t_{\text{Read}}) = 1 - \kappa$; $\eta(t_{\text{Random}}) = \alpha$; $\eta(t_{\text{Sequential}}) = 1 - \alpha$; $\eta(t_{\text{Small}}) = \beta$; and $\eta(t_{\text{Large}}) = 1 - \beta$.

The marking of place $pResource(R)$ (for both read or write activity) may denote the adopted technology. For instance, for traditional SSDs (SATA interface), the marking place *pResource* is 1, as only one operation at a time is carried out. Concerning SSDs-NVMe, *pResource* assumes the number of threads concurrently processing I/O requests (generally 8).

The proposed multi-storage model expands the model for multiple devices:

$$EC_h = \left(\sum_{d=0}^{n} \eta(tForward_d) \cdot EP_d \right) \cdot TH_h \cdot \text{time} \tag{2.8}$$

where the immediate transitions $tForward_d$ denote a request redirection to storage d.

Validation The model proposed by Borba et al. was validated using controlled experiments with the Fio benchmarking tool, which generated synthetic I/O workloads to measure and correlate storage system performance and energy consumption across varying request sizes, access patterns, and read/write ratios. Model estimates consistently fell within the 95% confidence intervals of observed system metrics. This statistical consistency indicates that the model's predictions are not significantly different from real-world values, supporting its applicability for performance and energy analysis in large-scale storage systems.

Limitations The authors acknowledge that a large number of devices significantly increases modeling complexity due to state space size explosion and recommend simulation as a viable workaround. Additionally, the authors acknowledge their focus on active energy states (not idle, standby states, or state transitions), treating them as delays between requests.

In a running server system, this approach could be adapted to create an accurate and fine-grained energy consumption estimation of a read/write workload on specific storage devices, albeit with limitations:

- Instead of needing to be estimated, (device-specific) throughput (TH) can be measured.
- An initial calibration run is necessary to experimentally determine the respective device-specific variables.
- In a multi-storage device server, the resulting state explosion may lead to significant calculation overhead, resulting also in higher energy consumption of the measurement itself.
- Instead of modelling transitions to a storage device as a function (as done in $\eta(tForward_d)$), device usage would actively need to be measured, which would essentially transform the multi-storage model into a simple addition of single-storage models. This would drastically reduce the number of total states, making calculations less demanding.

- Due to the authors not considering idle and standby states, a small, constant idle power consumption would need to be added to the model. This is especially important for accurate storage device power consumption modeling on idling or overprovisioned servers.

2.5.5 Network devices

Estimating the total power consumption of a network infrastructure requires a clear definition of system boundaries. Since most server clusters operate within larger, interconnected systems, a full assessment of network energy consumption (such as for CO_2 footprint calculations) is generally infeasible. This thesis limits the system boundary to the server itself, considering only internal network components, primarily the Network Interface Card (NIC). While this allows detailed modeling of NIC power usage, it excludes broader network activity, such as inter-node communication in multi-node clusters.

Although the overall energy consumption of a data center network could be estimated by including access, aggregation, and core switches, attributing this consumption to specific workloads remains highly challenging. This chapter therefore focuses on model-based methods for estimating NIC-level power as a proxy for server-side network energy usage.

2.5.5.1 NIC power consumption characteristics

While extensive research has analyzed the power consumption of network equipment like switches, routers, or gateways, NICs (especially non-wireless NICs) have not received as much attention. While several methods exist that modern NICs use to save power (e.g. PCIe Link power states and D-states, *Active State Power Management (ASPM)*, or *Energy Efficient Ethernet (EEE)*), there are no widely available mechanisms for fine-grained NIC power consumption estimation. As a consequence, NIC power can only be approximated based on the few available metrics.

Sohan et al.[41] measured and compared the power consumption of six 10 Gbps and four multiport 1 Gbps NICs at a fine-grained level. While they do not provide a method to estimate NIC energy consumption, they noted significant variation in power consumption between different NICs. Unfortunately, it cannot be ruled out that some results are cherry-picked: Solarflare NICs tend to dominate the introduced metrics, and a communications spokesperson is prominently credited with contact information. Regardless, some findings are consistent across manufacturers and align with other literature sources[99]. While these findings cannot directly contribute to a potential NIC power consumption estimation approach, they are relevant to understanding underlying mechanisms and assessing the relative importance of the NIC compared to other server components.

- **Idle Power**
  - The measured NICs showed power consumption between 5–20W.
  - Link connection status had little effect on idle energy consumption.
  - Physical media influenced power consumption: CX4 models had the lowest power consumption due to the simple design of the CX4 interconnect.

This was followed by fiber models. Finally, Base-T models consumed significantly more power due to the signal processing components in the card.

- **Active Power**
  - There was very little difference in the power usage of an active NIC compared to an idle one. For all measured NICs, the difference in power usage was less than 1W.
  - Throughput performance varied widely, and no correlation between power usage and performance was observed.
  - Power consumption increased in correlation with the number of ports.

In 2012, Basmadjian et al.[42] modeled a NIC by separating NIC power consumption into idle mode and dynamic mode (as they did for their CPU and RAM models). If $P_{NIC_{idle}}$ is the power of the idle interface and $P_{NIC_{dynamic}}$ is the power when active, the total NIC energy consumption is given by:

$$E_{NIC} = P_{NIC_{idle}} T_{idle} + P_{NIC_{dynamic}} T_{dynamic} \tag{2.9}$$

where T_{idle} and $T_{dynamic}$ are the total idle and dynamic times, respectively. Consequently, the average power during period T is given by:

$$P_{\text{NIC}} = \frac{(T - T_{\text{dynamic}}) P_{NIC_{idle}} + P_{NIC_{dynamic}} T_{\text{dynamic}}}{T} \tag{2.10}$$

$$= P_{NIC_{idle}} + (P_{NIC_{dynamic}} - P_{NIC_{idle}})\rho \tag{2.11}$$

where $\rho = \frac{T_{\text{dynamic}}}{T}$ is the channel utilization. While this formula is only helpful when NIC idle and max power consumption are already known, it shows that NIC power consumption is assumed to rise linearly with channel utilization.

Arjona Aroca et al.[48] modeled NIC efficiency based on their previous measurements. They found that NIC efficiencies for both sending and receiving are almost linear with the transfer rate and deduced a linear dependency on the network throughput.

In 2016, De Maio et al.[100] proposed a network energy consumption model for node-to-node transfers to estimate the total energy consumption required for virtual machine migration. Unfortunately, their model does not specifically handle NIC energy consumption, opting to model the entire node's energy consumption instead.

Another approach is presented by Dargie and Wen[101], who used stochastic modeling to examine the relationship between the utilization of a NIC and its power consumption, expressing these quantities as random variables or processes. They used curve fitting to determine the relationship between utilization and measured energy consumption of their specific NIC, after creating a dataset using a SPECpower benchmark. They assumed a uniformly distributed bandwidth utilization in the interval [0,125] MBps. Interestingly, their model showed only a slight effect of utilization on the predicted power consumption, mirroring the findings of Sohan et al.

The most recent NIC power model was proposed by Baneshi et al.[43] in 2024, analyzing per-application energy consumption. The authors noted that NIC idle power consumption may contribute up to 90% of the total NIC energy consumption. They proposed the following model for per-application NIC power consumption:

$$E_{\text{active}} = \sum_i \left(BW_i \cdot T_{\text{interval}} \cdot \frac{P_{\max} - P_{\text{idle}}}{BW_{\text{aggregated}}} \right) \tag{2.12}$$

$$E_{\text{idle}} = \sum_i \left(BW_i \cdot T_{\text{interval}} \cdot \frac{P_{\text{idle}}}{BW_{\text{used}}} \right) \tag{2.13}$$

where BW_i is the bandwidth of application i, $BW_{\text{aggregated}}$ is the aggregate bandwidth of both the uplink and downlink of the NIC, and BW_{used} is the used bandwidth of links (uplinks, downlinks, or both). The authors combined these formulae with power figures of their specific use case (total network power consumption in a fog computing scenario), which unfortunately are not applicable in the context of this thesis. Regardless, while these formulae cannot be used to estimate the maximum and idle NIC power, they can be applied irrespective of server specifications if idle and maximum NIC power consumption are known.

In contrast to the formulae presented by Basmadjian and Arjona Aroca, these formulae not only account for time intervals but also bandwidth used. As a result, the formulae presented by Baneshi et al. represent the current best approach to estimate NIC power consumption, even though this estimation still requires an initial guess of the idle and maximum power consumption. The author of this thesis is not aware of a more detailed formula currently available. A generalizable formula for estimating overall NIC power consumption is unlikely to exist due to the vast variety of NICs and the significant differences between manufacturers (as found by Sohan et al.).

2.5.6 Other devices

While much of the research on server energy consumption focuses on primary components such as the CPU, memory, storage, and network interfaces, a complete energy model must also account for additional hardware subsystems. These include the motherboard, power supply unit (PSU), system fans, and potentially other auxiliary devices. Though their individual energy consumption may appear minor compared to high-performance components, they collectively contribute a non-negligible share to the overall server power draw.

Despite their importance, these secondary components have received limited attention in energy modeling literature. In most cases, they are either omitted or treated as part of the residual power not attributable to the main computational subsystems.

The motherboard, for instance, includes voltage regulators, chipset logic, and peripheral interfaces. While these components may not be individually monitored, the Baseboard Management Controller (BMC) (see § 2.3.2.1) may expose aggregated power telemetry via vendor-specific sensors or interfaces like IPMI or Redfish. However, this level of detail varies greatly across hardware platforms and is seldom fine-grained enough for component-level attribution.

2.5.6.1 PSU

The power supply unit (PSU) is another often-overlooked consumer. When modeling the power usage of a PSU itself (distinct from the power it delivers to other components), the key factor is its conversion efficiency. PSUs consume more power than they deliver due to losses during AC–DC transformation and voltage regulation. According to Basmadjian et al.[42], the power consumed by the PSU can be approximated for various scenarios:

If the monitoring system provides information at the PSU level, its power consumption is given by

$$P_{PSU} = \frac{measuredPower \cdot (100 - e)}{100} \tag{2.14}$$

where e is the efficiency of the PSU.

If the monitoring system provides information at the server level, the power consumption of any of the n PSUs is given by the following formula, assuming that measured power is evenly distributed among PSUs:

$$P_{PSU} = \frac{\frac{measuredPower}{n} \cdot (100 - e)}{100} \tag{2.15}$$

If the monitoring system does not provide PSU power consumption, it can be deduced by

$$P_{PSU} = \frac{P_{Mainboard} + P_{Fans}}{n \cdot e} \cdot 100 - \frac{P_{Mainboard} + P_{Fans}}{n} \tag{2.16}$$

2.5.6.2 Fans

Cooling systems, particularly fans, also represent a meaningful share of the total energy budget. Most servers employ multiple fans controlled via Pulse-Width Modulation (PWM). While the BMC or operating system tools (e.g. `lm-sensors`) often report fan RPM or PWM duty cycle, actual fan power consumption is rarely exposed directly. Furthermore, RPM alone is insufficient to estimate power accurately, as fan power depends on physical factors such as the fan diameter, pressure increase, or air flow delivered[42]:

$$P_{Fan} = d_p \cdot q = \frac{F}{A} \cdot \frac{V}{t} = \frac{F \cdot d}{t} \tag{2.17}$$

where p_d denotes total pressure increase of the fan (Pa or N/m^2), q denotes the air volume flow (m^3/s), F denotes force (N), A denotes fan area (m^2), V denotes volume (m^3), and t denotes time (seconds).

Based on observations, F is proportional to the square of RPM. This can be combined with formula 2.17:

$$P_{Fan} = \frac{c \cdot RPM^2 \cdot d}{3600} \tag{2.18}$$

where for each individual fan, $c = \frac{3600 \cdot P_{Max}}{RPM_{Max}{}^2 \cdot d}$ remains constant.

Unfortunately, with the wide variety of fans in servers (especially with fan size restrictions due to server heights), these formulae are only helpful when paired with more detailed information on fan characteristics like maximum RPM and power.

While these can more reasonably be assumed, this remains a rough estimation at best.

2.5.6.3 Attribution of secondary component power consumption to individual workloads

Despite these measurement limitations, estimating the energy consumption of secondary components is often less critical for attributing energy to workloads. This is because components like fans, mainboards, and PSUs primarily support the operation of primary subsystems. Their power consumption scales with the activity level of CPU, memory, disk, and networking devices: more computation leads to higher heat dissipation, increased power delivery, and thus greater fan and PSU activity.

Consequently, if the total server power consumption is known (for example, via wall power monitoring or BMC/Redfish readings) the residual power (i.e., total power minus the sum of measured CPU, RAM, disk, and network power) can be reasonably attributed to power delivery and thermal management subsystems. This residual can then be proportionally distributed among active workloads based on the power consumption of the primary components they utilize. In this context, fine-grained modeling of secondary components becomes unnecessary for workload attribution, as their energy use correlates closely with that of the primary subsystems they support.

2.5.7 Issues with model-based power estimation techniques

This thesis pursues two inherently conflicting objectives: achieving high-resolution, accurate energy consumption measurements while simultaneously developing a solution that remains broadly applicable across heterogeneous server environments without requiring extensive manual calibration or the manual input of complex device-specific information. Striking a balance between these goals is particularly challenging in the context of model-based energy estimation for devices that do not expose power telemetry data. Due to the wide variability among devices for a variety of factors, energy consumption models must necessarily abstract away much of the underlying complexity. Developing a model that is simultaneously fine-grained, highly accurate, and universally applicable across different technologies is, in practice, an unattainable goal.

As a result, any model integrated into a general-purpose energy estimation framework must err on the side of relative simplicity to preserve generality. While this approach diminishes the precision of device-specific energy attribution, it remains valuable for broader optimization tasks. For instance, autoscaling mechanisms, load balancers, and schedulers can still benefit significantly from approximate energy profiles. Likewise, cluster administrators aiming to improve energy efficiency holistically, or developers seeking to optimize their workloads, can gain useful directional insights even from coarse-grained models.

However, this simplicity imposes significant limitations for use cases that require device-specific energy optimization. General-purpose models are ill-suited for evaluating the energy efficiency of different device types, testing firmware-level adjustments, or validating the impact of power-saving features such as low-power states. In such scenarios, the model's abstraction may not just be insufficient, but actively misleading.

In the context of this thesis, this limitation is considered acceptable. The overarching objective is to facilitate scalable and portable energy estimation mechanisms for containerized environments, not to provide a diagnostic tool for hardware-level energy analysis. Nonetheless, this constraint should be kept in mind when interpreting the results and assessing their suitability for device-centric evaluation tasks.

2.6 Power Modeling based on Machine Learning Algorithms

In a taxonomy of power consumption modeling approaches, Lin et al.[2] analyze various machine learning-based power models in current literature, categorizing them into supervised, unsupervised, and reinforcement learning. A detailed reiteration of this taxonomy (as well as a methodological overview of machine learning and neural networks) is omitted here.

In the context of this thesis, machine learning-based approaches are not considered for the following reasons:

- The author was unable to identify a promising and reliable approach that is sufficiently generalizable to function across a wide range of server configurations. Likewise, no component-level models were found that met these criteria. While it is certainly possible to train machine learning models to estimate energy consumption for specific server setups with high accuracy, the aim of this thesis is to provide generalizable estimation methods applicable to varied systems.

- Machine learning fundamentally relies on large datasets that are both highly accurate and granular, ideally matching the quality expectations of the resulting model. As discussed in previous sections, such high-quality training data is rarely available in the domain of fine-grained power measurement. When such datasets do exist, they typically reflect highly specific hardware and workload configurations, making them unsuitable for generalization. Although it would be theoretically possible to generate a large dataset by systematically benchmarking thousands of CPUs, memory modules, GPUs, storage, and network devices across millions of configurations, such an undertaking is not practically feasible. Furthermore, any such dataset would require ongoing expansion to remain representative of new hardware generations.

- Finally, many of the technical implementation details underlying key telemetry features remain proprietary or undocumented. A notable example is the RAPL interface, whose internal workings are not publicly disclosed. At the same time, existing RAPL metrics already offer abstracted energy readings suitable for direct integration into power estimation tools, eliminating the need for an intermediate machine learning-based step.

In theory, machine learning-based power estimation models hold significant promise and may one day be realized. Such models could leverage complex, nonlinear relationships between hardware components and workloads, relationships that are inherently difficult or even impossible to capture through traditional analytical models. The predictive and adaptive capabilities of machine learning offer the potential for highly accurate, fine-grained estimations across a broad range of configurations. However, as of today, the development of such a comprehensive and generalizable

model remains out of reach. Realizing this vision would require extensive collaboration between original equipment manufacturers, cloud providers, data center operators, and research institutions to generate, standardize, and share high-quality telemetry data across diverse hardware and workload scenarios. Given the role of cloud computing in the current economic landscape, where proprietary knowledge and performance optimization constitute a competitive advantage at the corporate, national, and geopolitical levels, such broad cooperation appears unlikely. Consequently, machine learning remains a promising but currently impractical direction for universal server power modeling.

2.7 Component-specific summaries

This section offers practical guidelines for measuring or estimating the energy consumption of individual hardware components. While earlier chapters focused on theory and research, the focus here is on implementation: what works best, what challenges to expect, and how to improve accuracy with minimal effort. For each component, the most accurate method is highlighted, along with alternatives and fallback options. Simple improvements like entering datasheet values or running calibration workloads are discussed where relevant.

2.7.1 CPU

The most accurate and widely adopted method for measuring CPU energy consumption is Intel's RAPL interface. It offers high temporal resolution, low overhead, and requires no external hardware. Among the available interfaces, `perf-events` is generally recommended due to its balance between usability, performance, and access control. The `powercap` interface offers simpler integration via `sysfs`, though with some limitations in domain structure and overflow handling. MSR access is discouraged due to complexity and privilege requirements. eBPF-based methods are powerful and used in advanced tools, but introduce high development complexity, kernel dependency, and lower portability.

On AMD systems, the `amd_energy` driver offers a RAPL-compatible interface, but it exposes fewer domains (e.g. no DRAM) and is generally less feature-rich than Intel's implementation.

Despite some limitations (such as non-atomic register updates, idle power inaccuracies, and the absence of timestamps), RAPL is considered sufficiently accurate for both research and production use. Its drawbacks can often be mitigated through careful sampling strategies (e.g. overflow-safe polling intervals, timestamp alignment) and correction techniques for overflow and measurement jitter.

ACPI, while historically relevant, does not expose real-time power data and is unsuitable for precise energy measurement. Although some theoretical estimation based on P-states is possible, it is coarse-grained and impractical for modern CPUs with dynamic frequency scaling.

If RAPL is unavailable, statistical models based on utilization, frequency, and other metrics may be used. However, these require prior calibration or hardware-specific profiling. Without access to idle or peak power values, such models become highly unreliable due to architectural variability. In such cases, estimation accuracy can be

modestly improved by inserting static power values from processor datasheets or using fixed coefficients for known CPU families.

2.7.1.1 Container-level implications

RAPL's granularity and domain separation make it suitable for correlating CPU energy usage with container activity, allowing for reasonably accurate attribution when combined with CPU usage metrics (e.g. cgroup CPU accounting or eBPF). In contrast, model-based estimations or ACPI-derived values are too coarse and lack temporal resolution, limiting their use to static or linear power distribution based on workload share, which is insufficient for fine-grained or bursty container workloads.

2.7.2 Memory

Memory power consumption can be measured using the DRAM domain exposed by Intel RAPL on supported server-grade processors. When available, this provides low-overhead, fine-grained energy telemetry integrated with other CPU domains. However, DRAM measurement accuracy depends heavily on processor architecture. It is generally reliable for Haswell-generation CPUs, but later architectures may exhibit a constant power offset or measurement inaccuracies due to off-DIMM voltage regulators and evolving memory subsystems.

If RAPL DRAM telemetry is unavailable or deemed unreliable, no equivalent in-band method exists. In such cases, estimation must rely on model-based approaches. Many models in the literature attempt to correlate memory power with usage, memory access frequency, or cache behavior, but they are not generalizable across systems. Most require prior calibration using known idle and peak memory power figures, which are rarely available in practice. Without these, estimation accuracy remains low. Manual insertion of idle and active power values from vendor datasheets can slightly improve results, but still yields only coarse-grained estimates.

2.7.2.1 Container-level implications

The RAPL DRAM domain, when accurate, allows correlation between energy consumption and workload-level memory metrics such as usage or memory bandwidth. This enables container-level attribution if per-container memory activity is available. Without RAPL, model-based estimates only support static or proportional energy attribution based on usage share, which is insufficient for capturing the energy impact of memory-intensive or bursty workloads.

2.7.3 GPU

Accurate GPU power measurement remains a challenge in containerized environments. The most accessible solution is NVIDIA's NVML interface (e.g. via nvidia-smi), which exposes power metrics through on-board sensors. While widely used, NVML suffers from sampling delays, averaging artifacts, and limited temporal resolution, especially during transient workloads. Nevertheless, it offers acceptable accuracy for steady-state measurements and is supported across many data center deployments.

Alternative tools, such as AccelWattch and FinGraV, provide finer temporal granularity and more precise modeling but are either architecture-specific or tightly coupled to particular hardware (e.g. AMD MI300X). Hardware-based solutions like PowerSensor3 achieve excellent accuracy at high sampling rates but are cost-prohibitive and impractical for large-scale deployment. No general-purpose, software-only solution currently matches the accuracy and portability of CPU-side tools like RAPL.

2.7.3.1 Container-level implications

GPU power attribution in Kubernetes is limited by the granularity and accuracy of current tools. While NVML can be queried from within containers or sidecars, it does not natively support multi-tenant attribution, and virtualization layers (e.g. vGPU, MIG) complicate per-container visibility. Accurate container-level GPU energy tracking remains an open problem, requiring either architectural integration (e.g. with MIG-aware scheduling) or improved temporal sampling. As such, GPU measurements are currently only viable for coarse-grained, workload-level profiling, not fine-grained container energy attribution.

2.7.4 Storage devices

Storage device energy consumption is typically estimated rather than measured. Unlike CPUs or GPUs, storage devices lack onboard power sensors, and BMC-based per-device readings are generally unavailable, especially when using backplanes, RAID controllers, or SATA interfaces. Consequently, power usage is inferred from device metrics and modeled behavior.

Telemetry is only available for specific device types. For NVMe drives, `nvme-cli` exposes detailed metrics such as supported power states, current power state, idle/active power ratings, and temperature. However, these are not available for SATA SSDs or HDDs. `smartctl` provides vendor-specific SMART data (e.g. temperature, power-on hours, wear) if available, but energy-related insights are limited. Standard Linux tools (`iostat`, `sar`, `/proc/diskstats`, etc.) expose generic performance counters such as IOPS, throughput, queue length, and utilization, which can support rough estimation.

Various model-based approaches estimate power using activity-based metrics (e.g. read/write rates or interface speed), often requiring idle and active power values from datasheets. These models are only accurate when tailored to specific hardware. No general-purpose estimator exists for unknown or heterogeneous storage types without prior calibration.

2.7.4.1 Container-level implications

Because disks are shared resources and per-container telemetry is unavailable, energy attribution must rely on proportional estimation using observable metrics like I/O volume or latency. This approach works for long-lived workloads but lacks the granularity to capture energy dynamics of bursty or short-lived container activity. Accurate container-level attribution remains infeasible for SATA SSDs and HDDs and is only marginally better for NVMe devices, assuming access to detailed device metrics.

2.7.5 Network devices

NIC power consumption cannot be measured directly via software. Although many cards support PCIe power states (e.g. D0–D3), these states only approximately correlate with actual power draw and are not sufficient for energy estimation. Furthermore, NICs lack onboard power sensors, and BMC-based per-device readings are generally unavailable. As such, NIC energy consumption must be estimated using model-based approaches.

Various research models estimate NIC power using idle and dynamic components. The most promising approach, proposed by Baneshi et al., linearly scales NIC power with bandwidth utilization, assuming idle and maximum power values are known. While earlier models correlate energy use with channel utilization or throughput, they often oversimplify or lack generalizability. Real-world measurements show minimal power variation between idle and active states (often <1W difference), with idle power dominating overall NIC energy use. Estimates can be improved slightly by incorporating known idle and peak wattage from datasheets, but generalization across different NICs remains unreliable due to architectural and vendor variability. In the absence of these values, the only remaining option is to guess these values based on NIC PHY medium.

Telemetry support is limited: tools like `ethtool` expose link speed and status but do not report power. No standard Linux tool provides direct NIC energy metrics, and throughput-based estimators must rely on indirect metrics like bytes transmitted per interval.

2.7.5.1 Container-level implications

Because NICs are shared across containers and lack per-container telemetry, only indirect attribution is possible. Energy consumption can be distributed proportionally based on container-level bandwidth usage (e.g. via cgroup network statistics), assuming idle and peak NIC power are known. However, the minimal dynamic variation in NIC power limits the usefulness of fine-grained attribution. In practice, NIC power is best modeled as a mostly static overhead, with marginal gains from utilization-based scaling.

2.7.6 Other Devices

Secondary components such as the motherboard, PSU, and fans contribute a non-trivial share to total server power consumption but are rarely modeled with precision. These devices typically lack direct power telemetry, and their energy use is either approximated or inferred indirectly.

PSU losses can be estimated from efficiency ratings if total input or output power is known. Fan power is difficult to measure and depends on physical factors like airflow and pressure; at best, it can be roughly estimated using RPM and vendor data. The motherboard and onboard controllers (e.g. voltage regulators, chipset) are usually modeled as part of residual power.

2.7.6.1 Best-practice approach

If system-level power data is available (e.g. via IPMI or Redfish), the difference between total server power and known component estimates can be treated as residual

power. This residual can be linearly distributed across containers based on the finer-grained power estimation of the CPU, assuming secondary device power scales with primary component power consumption.

2.7.6.2 Container-level implications

Because these components do not map directly to container usage, their energy must be attributed indirectly. Linear distribution based on known, container-attributed metrics (e.g. CPU time or workload duration) is a practical, though imprecise, fall-back for ensuring full power accounting in containerized environments.

Chapter 3

Attributing Power Consumption to Containerized Workloads

3.1 Introduction and Context

While the previous chapter focused on system-level and component-level power measurement and estimation, this chapter shifts focus to an equally complex task: attributing measured server power consumption to the individual containers or workloads responsible for it.

Attributing energy consumption in this context is inherently difficult due to multi-tenant, multi-layered workloads across multiple CPU cores and devices, as well as temporal granularity mismatch issues. Consequently, direct one-to-one mapping of energy consumption to workloads is generally not possible.

Nonetheless, various techniques have emerged to approach this problem. The goal is to create an accurate and fair approximation of how much energy a given container or process is responsible for at any point in time. This chapter provides a conceptual foundation for these techniques. The subsequent **Chapter 4** will examine how selected tools implement these ideas in practice. While some implementation aspects will be referenced for illustration, this chapter focuses on general methodologies, not tool-specific behavior.

3.2 Power Attribution Methodology

3.2.1 The Central Idea Behind Power Attribution

At its core, the concept of power attribution is simple: a task should be held accountable for the energy consumed by the resources it actively uses. If a task occupies the CPU for a given period, it is attributed the energy consumed by the CPU during that time. By summing the energy usage of all tasks belonging to a container, one can estimate the total energy consumption of that container. Since energy is the integral of power over time, the average power consumption of a container can be calculated by dividing its attributed energy by the total duration of interest. Depending on the use case, either energy (in joules) or power (in watts) may provide more meaningful insight. Energy is often used to quantify cost or carbon footprint, while power helps identify peak loads and inefficiencies.

While this model appears intuitive, its implementation in real systems is far from trivial. One major complication stems from the intricacies of multitasking on modern systems, which is discussed in § 3.2.2. § 3.2.3 examines and compares different utilization tracking mechanisms in Linux and Kubernetes. As a result of the fine-grained temporal control of multitasking, another major challenge is temporal granularity. Power consumption is typically sampled at much coarser intervals than kernel resource usage statistics. These mismatched update rates and resolutions must be reconciled to build meaningful correlations. This issue is elaborated in § 3.2.4.

Consequently, power attribution becomes a complex algorithmic process, involving summation, weighting, and interpolation across multiple metrics. It must strike a balance between data availability and estimation accuracy. A perfectly accurate system is not feasible, especially in heterogeneous or production-grade environments. Limitations and accuracy trade-offs are further discussed in § 3.2.5. Finally, § 3.3 discusses the different philosophies of various attribution models to account for different key demographics.

Despite these difficulties, power attribution serves a critical role in understanding container behavior. If applied consistently across all containers and system resources, it can uncover the dynamic patterns of energy usage within a server. This insight forms a foundational building block for cluster-level energy optimization. Administrators or automated systems can use this data to analyze the effect of configuration changes, improve workload scheduling, or optimize performance-per-watt, whether during runtime or post-execution.

3.2.2 A Short Recap of Linux Multitasking and Execution Units

Linux is a multitasking operating system that enables multiple programs to run concurrently by managing how processor time is divided among tasks. This capability is central to container-based computing and directly impacts how workload activity is linked to energy consumption.

Multitasking in Linux operates on two levels: time-sharing on a single core and true parallel execution across multiple cores. On a single-core system, the kernel scheduler rapidly switches between tasks by allocating short time slices, creating the illusion of parallelism. On multi-core systems, tasks can run simultaneously on different cores, increasing throughput but also complicating the task of correlating resource usage with measured power consumption.

At the kernel level, the smallest unit of execution is a *task*. This term covers both user-space processes and threads, which the kernel treats uniformly in terms of scheduling and resource accounting. Each task is represented by a `task_struct`, which tracks its state, scheduling data, and resource usage.

A *process* is typically a task with its own address space. Threads, by contrast, share memory with their parent process but are scheduled independently. As a result, a multi-threaded program or container may generate several concurrent tasks, potentially running across multiple cores. These tasks are indistinguishable from processes in kernel metrics, which complicates aggregation unless care is taken to associate related threads correctly.

In containerized environments, tasks belonging to the same container are grouped

using Linux control groups (cgroups) and namespaces. These mechanisms allow the kernel to apply limits and collect resource usage statistics at the container level, making them central to energy attribution in Kubernetes-based systems.

3.2.3 Resource Utilization Tracking in Linux and Kubernetes

In modern Linux-based systems, particularly within Kubernetes environments, multiple methods exist to track resource utilization [30–33, 102, 103]. These methods vary significantly in terms of temporal granularity, scope, and origin. While they often expose overlapping information, their internal mechanisms differ, leading to trade-offs in precision, resolution, and suitability for certain use cases such as energy attribution.

3.2.3.1 CPU Utilization Tracking

- **/proc/stat:** A global, cumulative snapshot of CPU activity since boot. It records jiffies spent in user, system, idle, and iowait modes. Temporal resolution is high, but data is coarse and not process- or cgroup-specific.
- **/proc/<pid>:** Provides per-task CPU statistics including time spent in user and kernel mode. Offers fine-grained tracking on a per-process level but must be polled manually at high frequency to detect short-lived changes. Contains information about task container and namespace.
- **cgroups:** Tracks cumulative CPU usage in nanoseconds per cgroup. In Kubernetes, each container runs in its own cgroup, enabling container-level usage attribution. Granularity is high, and this is a foundational metric for tools like KEPLER and cAdvisor.
- **eBPF:** eBPF enables near-real-time tracking of per-task CPU cycles and execution, allowing correlation of resource usage to kernel events (e.g. context switches). It is especially valuable when precise attribution to short-lived tasks or containers is required.
- **Hybrid tools:** Many tools provide aggregated metrics and statistics based on the aforementioned methods. While user-friendly, these usually offer lower temporal precision, but may be useful in some instances. **cAdvisor:** collects and aggregates CPU usage per container by reading from cgroups. While widely used, its default update interval is coarse. Data is sampled and averaged, which limits its use in high-resolution analysis. **metrics-server (metrics.k8s.io):** exposes aggregated CPU usage via the Kubernetes API. It pulls metrics from Kubelet (which relies on cAdvisor) and is updated approximately every 15 seconds. Not suitable for precise or historical analysis.

3.2.3.2 Memory Utilization Tracking

- **/proc/meminfo:** Provides a system-wide view of memory usage but lacks per-task or per-container resolution.
- **/proc/<pid>/status:** Exposes memory-related counters for each process (e.g. RSS, PSS, virtual set size). Temporal granularity is fine but requires frequent polling.

- **cgroups (memory):** Records memory usage for groups of processes. `memory.usage_in_bytes` shows current memory usage per cgroup, allowing container-level tracking. High granularity and reliability, frequently used in both monitoring and enforcement.

- **cAdvisor and metrics-server:** As with CPU, memory stats are aggregated from cgroup data. These APIs offer lower resolution and no historical data.

3.2.3.3 Disk I/O Utilization Tracking

- **/proc/<pid>/io:** Tracks per-process I/O activity (bytes read/written, syscall counts). Useful for attributing I/O behavior, but coarse in how it correlates to actual disk access timing.

- **cgroups-v1 (blkio) / cgroups-v2 (io):** Reports aggregated I/O stats per cgroup (bytes, ops, per-device). Allows container-level attribution. Granularity depends on polling rate and support by the underlying I/O subsystem.

- **eBPF (tracepoints, kprobes):** Enables real-time tracing of block I/O syscalls, bio submission, and completion.

3.2.3.4 Network I/O Utilization Tracking

- **/proc/net/dev:** Shows network statistics per interface. Updated continuously, but lacks process/container granularity.

- **cgroups-v1 (net_cls, net_prio):** Used to mark packets with cgroup IDs, enabling traffic shaping and classification. Attribution is possible if paired with packet monitoring tools, but rarely used directly. While there is no direct equivalent in **cgroups-v2**, support was added in `iptables` to allow BPF filters that hook on cgroup v2 pathnames to control network traffic on a per-cgroup basis.

- **eBPF:** Allows tracing of network activity at various points in the stack (packet ingress, egress, socket calls). Offers very high granularity and can attribute traffic to specific containers.

3.2.3.5 eBPF-based Collection of Utilization Metrics

The extended Berkeley Packet Filter (eBPF) is a Linux kernel subsystem that allows the safe execution of user-defined programs within the kernel without modifying kernel source code or loading custom modules. Originally developed for low-level network packet filtering, eBPF has evolved into a general-purpose observability framework that can trace and monitor system events with high precision and minimal overhead. eBPF can be used to dynamically attach probes to kernel events such as context switches, system calls, I/O events, and tracepoints. In the context of system monitoring, this enables the collection of fine-grained utilization metrics, including CPU usage per process, memory allocations, and I/O activity, without modifying the monitored application. These probes run within the kernel and populate BPF maps, which can then be accessed by user-space tools to aggregate or export metrics.

Compared to traditional monitoring approaches such as reading from `/proc`, eBPF offers several key advantages. First, it supports high temporal resolution, enabling near real-time tracking of events. Second, it avoids the need for intrusive instrumentation or static tracepoints, making it suitable for black-box applications. Finally, its dynamic and event-driven nature reduces performance overhead by eliminating polling. As a consequence, eBPF has often been used for utilization monitoring: KEPLER uses eBPF to monitor CPU cycles and task scheduling events, enabling accurate attribution of resource usage to short-lived or highly dynamic workloads. It complements cgroup and perf-based metrics, allowing power attribution models to track containers that would otherwise be indistinguishable using standard polling-based methods.

As demonstrated by Cassagnes et al. [34], eBPF currently represents the best practice for non-intrusive, low-overhead, and high-resolution utilization monitoring on Linux systems. Its ability to gather container- or process-level metrics in production environments makes it uniquely well-suited for accurate correlation with system-wide power measurements.

3.2.3.6 Performance Counters and `perf`-based Monitoring

Modern processors expose hardware-level performance counters (PMCs) that can be used to obtain precise measurements of internal execution characteristics. These counters are accessible via tools such as `perf`, and include metrics such as retired instructions, CPU cycles, cache misses, branch mispredictions, and stalled cycles. Unlike traditional utilization metrics, which measure time spent in various CPU states, PMCs offer insight into how effectively the processor is executing instructions.

A particularly relevant metric is *instructions per cycle* (IPC), which quantifies how much useful work is being done per clock cycle. An IPC close to the CPU's architectural maximum indicates efficient execution, while lower values often signal bottlenecks such as memory stalls. As shown by Gregg[35], a low IPC may reveal that the processor is heavily stalled, even when CPU utilization appears high.

These metrics provide a powerful alternative for workload analysis and energy estimation. For instance, instruction counts can be used to normalize energy usage per task, enabling attribution models that go beyond time-based utilization.

However, access to PMCs is not always guaranteed. In virtualized environments and some container runtimes, performance counters may be inaccessible or imprecise due to hypervisor restrictions. Moreover, interpreting raw PMC values requires architectural knowledge and hardware-specific calibration.

3.2.3.7 Comparative Summary

3.2.4 Temporal Granularity and Measurement Resolution

To correlate CPU usage with power consumption, time must be considered at an appropriate granularity. The Linux kernel tracks CPU usage at the level of scheduler ticks, which are driven by a system-wide timer interrupt configured via `CONFIG_HZ`. Typical values range from 250 to 1000 Hz, meaning time slices of 4 to 1 milliseconds, respectively. These ticks, or *jiffies*, represent the smallest scheduling time unit and are used to increment counters such as `utime` and `stime` for each task.

Source	Granularity	Scope	Notes
/proc/stat	Medium	Global	Jiffy-based, coarse
/proc/<pid>/stat	High	Per-process	Fine-grained, must poll manually
cgroups	High	Per-cgroup	Foundation for container metrics
cAdvisor	Medium-Low	Per-container	Aggregated from cgroups, limited rate
eBPF	Very High	Per-task, system-wide	Real-time, customizable, low overhead
perf/PMCs	Very High	Per-task, core-level	Tracks cycles, instructions, stalls

TABLE 3.1: Comparison of resource usage tracking mechanisms

More modern interfaces (such as cgroup v2's `cpu.stat`) provide higher-resolution timestamps, often in nanoseconds, depending on the kernel version and configuration.

In contrast, power measurement tools generally operate at coarser time resolutions. Intel RAPL, for example, may expose updates every few milliseconds to hundreds of milliseconds, while BMC- or IPMI-based readings typically update once per second or slower. As a result, power attribution techniques must reconcile the high-frequency task activity data with lower-frequency power measurements, often through aggregation or interpolation over common time intervals.

A clear understanding of these execution and timing units is essential for building reliable power attribution models. These concepts underpin all subsequent steps, including metric fusion, resource accounting, and workload-level aggregation.

3.2.5 Challenges

System monitoring and the attribution of power metrics based on system (and component) utilization metrics introduce several challenges that need to be addressed by a power attribution methodology. Some of these represent natural trade-offs that an architect needs to be aware of, while others pose issues that simply cannot be circumvented without major drawbacks that cannot be solved with a suitable architecture.

3.2.5.1 Temporal Granularity and Synchronization

A central challenge in power attribution is the mismatch in temporal granularity between system and power metrics. High-resolution sources, such as eBPF-based monitoring, can distinguish variations within individual CPU time slices. In contrast, coarse-grained power metrics (such as IPMI) often update only once per second, rendering them unable to reflect fine-grained container activity. Metrics like RAPL fall in between, typically sampled at up to 1000 Hz but practically stable at around 50 Hz. Model-based estimators may match the granularity of their input metrics or, in simpler cases, use time-based assumptions with theoretically unlimited granularity.

These disparities make straightforward correlation difficult. While coarse metrics like IPMI provide broad system power data (including components invisible to fine-grained tools), they should not be interpolated to finer time scales, as doing so introduces artificial detail and potential misattribution. Instead, they are best treated as low-frequency anchors to validate or constrain high-resolution estimates. For example, summed RAPL readings can be compared to IPMI over aligned intervals, though their differing measurement scopes add complexity.

Another complication is temporal skew. Even metrics with similar frequencies are rarely sampled simultaneously, and some introduce unknown or variable delays. This misalignment creates ambiguity between observed utilization and corresponding power draw, particularly for short-lived or rapidly changing workloads. Naïve smoothing may reduce noise but also obscures meaningful transient behavior.

Effective attribution therefore requires more than just aligning timestamps. It demands awareness of each metric's origin, behavior, and limitations, and careful coordination to avoid erroneous correlations and preserve meaningful detail.

3.2.5.2 Challenges in CPU Metric Interpretation

CPU utilization is one of the most accessible and commonly used metrics to quantify processing activity on modern systems. It is widely reported by system monitoring tools such as `top`, `htop`, and cloud APIs, and is frequently used in both performance diagnostics and energy attribution models. However, despite its ubiquity, the interpretation of CPU utilization is far from straightforward, and in many contexts, it is misleading.

At its core, CPU utilization is a time-based metric that represents the proportion of time a CPU spends executing non-idle tasks. In Linux, this value is computed from counters in `/proc/stat` and reported in units of "jiffies". It distinguishes between various states (user, system, idle, I/O wait, interrupts) but ultimately expresses how long the CPU was busy, not how much useful work it performed[104].

A fundamental limitation is that CPU utilization conflates time with effort. Not all CPU time is equally productive: some cycles may execute complex, compute-intensive instructions, while others may stall waiting for memory I/O. Modern CPUs are frequently memory-bound due to the growing performance gap between processor speed and DRAM latency. As a result, a high CPU utilization value may indicate that the processor was merely stalled, not that it was the performance bottleneck[35].

These nuances have direct consequences for energy attribution. When energy models allocate power proportionally to CPU utilization, they assume a linear relationship between time and energy. However, power consumption depends heavily on the instruction mix, CPU frequency scaling, Turbo Boost, and simultaneous multithreading. In such environments, identical utilization values across different processes or intervals may reflect vastly different energy profiles.

A more accurate alternative is to use hardware performance counters (PMCs), which track low-level metrics such as instructions retired, cache misses, and stalled cycles. For example, the "instructions per cycle" (IPC) value provides insight into how effectively the CPU executes work during its active time. An IPC significantly below the processor's theoretical maximum often indicates a memory-bound workload, while

high IPC values suggest instruction-bound behavior. Tools like `perf` or `tiptop` can expose such metrics, though their use may be restricted in virtualized environments.

In summary, CPU utilization should be treated with caution, especially in the context of energy-aware scheduling and workload attribution. As Cockcroft already argued in 2006, utilization as a metric is fundamentally broken[105]. Practitioners are advised to:

- Avoid assuming a linear relationship between CPU utilization and power consumption.
- Consider supplementing utilization metrics with performance counters (e.g. IPC, cycles, instructions) when available.
- Be mindful of the measurement interval and sampling effects in tools like *Scaphandre*.
- In energy models, explicitly account for idle power, and avoid assigning it solely to active processes.
- Prefer instruction-based metrics for finer granularity and better correlation with energy use.

3.2.5.3 Availability of Metrics

The availability of system and power metrics varies widely between platforms. While some systems offer high-resolution data, others may only expose coarse values or lack direct power data entirely. An effective attribution system should dynamically adapt to the metrics available, incorporating new sources (such as wall power meters) as they are added.

Ideally, such a system would also communicate the trade-offs involved, indicating how metric availability affects accuracy and granularity. This transparency ensures that attribution results are interpreted with appropriate context and helps guide improvements in monitoring fidelity.

Attribution in Multi-Tenant and Shared Environments

In multi-tenant systems, not all resources can be cleanly partitioned or measured with sufficient precision for container-level attribution. Some components are inherently shared, and their energy use cannot be isolated to individual workloads. Additionally, system-wide energy consumers like power supplies, cooling fans, and idle background services contribute to total power draw but are not tied to any specific container. Attribution models must account for these shared and unaccountable energy domains. Addressing these concerns requires careful modeling and philosophical choices about how to treat unassigned energy, which are further discussed in § 3.3.

Measurement Overhead

All monitoring systems inherently introduce some degree of overhead. While modern tools such as eBPF are designed to minimize this impact, they still consume CPU

cycles and memory bandwidth. Lightweight tools can reduce overhead without sacrificing data quality, but complete elimination is not possible.

Notably, the cost of monitoring increases with temporal resolution. Fine-grained metrics require higher sampling rates, more frequent data transfers, and additional processing effort. Since container-level power metrics typically do not require sub-second resolution, it is essential that high-resolution analysis and correlation occur as early as possible in the data pipeline. By aggregating and attributing power consumption close to the source, downstream systems can operate on compact, coarse-grained results, reducing both computational and storage overhead while preserving attribution accuracy.

Support for Evolving Models

As hardware platforms and research in power estimation continue to evolve, new measurement interfaces and modeling approaches are regularly introduced. These may offer improved accuracy, reduced overhead, or better coverage of previously unobservable components. To remain relevant and effective, container-level power attribution systems must be designed with adaptability in mind. A modular architecture enables the integration of new data sources or estimation models without reengineering the entire system. This flexibility ensures long-term maintainability and allows the system to benefit from ongoing advancements in energy modeling and monitoring infrastructure.

3.3 Attribution Philosophies

Attributing server power consumption to individual containers requires decisions that go beyond data collection. Some components are inherently shared, some workloads contribute system-level overhead, and some energy is consumed by idle hardware. The way these factors are treated reflects the underlying attribution philosophy. This section outlines three main approaches, each suitable for different goals and users.

3.3.1 Container-Centric Attribution

This model attributes energy solely based on the direct activity of containers, ignoring system services and shared infrastructure. Remaining resources are pooled and can be declared as system resources. This means that a container is not accountable for its own orchestration or energy wasted through system idling.

- **Advantages:** Isolates workload impact; consistent across system loads.
- **Limitations:** Understates real-world cost; excludes orchestration and idling overhead.
- **Suitable for:** Developers optimizing containerized applications.

Notably, container-centric attribution places a strong emphasis on individual containers and their respective energy consumption, striving to maintain relative consistency irrespective of overall cluster activity. While such granular insights can be valuable to developers, container-centric attribution typically does not represent the

primary practical use case of an energy monitoring system for Kubernetes containers. This is largely due to the container isolation principle, which usually restricts detailed visibility into broader system dynamics. Additionally, container-level optimization is often more effectively achieved through simpler CPU and memory metrics readily accessible via the container's own `/proc` filesystem. Hence, although technically feasible, container-centric energy attribution often remains primarily a theoretical or research-oriented concept rather than a widely implemented practical approach.

3.3.2 Shared-Cost Attribution

Here, all power consumption is distributed across active containers, either equally or proportionally to usage. As a consequence, a container is accountable for its own orchestration, its share of OS resources, and even energy wasted through system idling.

- **Advantages:** More accurately reflects total system cost.
- **Limitations:** Attribution fluctuates with container count; depends on arbitrary distribution logic.
- **Suitable for:** Cluster operators optimizing cluster orchestration.

3.3.3 Explicit Residual Modeling

Beyond the container-centric and shared-cost attribution models lies a more nuanced approach that explicitly incorporates the efficiency characteristics of server hardware. In this model, total power consumption is divided not only among containers and system services but also includes separate terms for idle power and high-utilization overhead. Idle power represents the baseline energy required to keep the system operational, even when no meaningful work is being performed. However, this value is difficult to isolate, as it often overlaps with low-level system activity such as kernel threads, background daemons, or monitoring agents.

At the other end of the spectrum, when utilization approaches system limits, energy efficiency typically degrades due to resource contention, frequent context switching, and thermal throttling[106]. These effects increase energy consumption without proportional performance gains. To account for these dynamics, this model introduces two residual domains (*idle waste* and *efficiency overhead*), which reflect conditions not attributable to any specific container. While this model is more complex, it enables more accurate assessment of workload behavior, infrastructure utilization, and waste, making it particularly valuable for research, performance engineering, and sustainability analysis.

Challenges in Measuring Residuals. Despite its advantages, implementing this model is non-trivial due to the difficulty of distinguishing idle consumption and overhead effects from general system resource usage:

Idle power estimation

- **Shared background activity:** Even in idle states, kernel tasks and system services introduce minimal but nonzero load, making it hard to define a "pure" idle baseline.
- **C-state transitions:** CPUs may briefly exit low-power states due to timers or interrupts, causing fluctuations even during apparent idleness.
- **Isolation difficulty:** In production or multi-tenant environments, isolating a server to a truly idle state is often impractical.

High-utilization overhead

- **Lack of a clear baseline:** There is no standard definition of "ideal" energy usage at full utilization, complicating quantification of overhead.
- **Architecture-specific behavior:** Overheads from cache contention, memory stalls, or I/O bottlenecks depend heavily on the workload and hardware architecture.

Due to their complexity and variability, high-utilization overhead effects are excluded from the scope of this thesis. This is a minor limitation, as assigning this energy to general *system* consumption remains a valid and conservative approach.

Practical Approach to Idle Estimation. In practice, idle consumption can be estimated pragmatically by recording power usage while no user workload is running. While this conflates pure idle consumption with background system activity, the trade-off is acceptable given its simplicity and reproducibility.

The resulting hybrid model separates power into three categories:

$$P_{\text{total}} = \sum P_{\text{container}} + P_{\text{system}} + P_{\text{idle}} \tag{3.1}$$

Residual power not attributed to container workloads is explicitly labeled as *system* or *idle* consumption. (In some literature, the term *static* is used in place of *idle*.)

- **Advantages:** Transparent; enables both container-level and infrastructure-level analysis.
- **Limitations:** Requires high-quality telemetry; boundaries between idle and system power are inherently fuzzy.
- **Best suited for:** Research, cluster optimization, and sustainability reporting.

In real-world scenarios, this model provides cluster operators with the foundation for quantitative infrastructure efficiency analysis. At the same time, developers benefit from a consistent, workload-centric power metric that reflects the true resource cost of their container, independent of the activity of co-located workloads.

3.3.4 Distinction Between CPU Idling and Process Idling

A CPU is considered **idle** when it has no runnable tasks. In this case, the Linux scheduler runs a special task called the *idle task* (PID 0), and the processor may enter a low-power idle state to save energy. The time spent in this state is what is reported

as CPU idle time. The *idle task* is not shown in `/proc` and similar interfaces because it is only internally used by the scheduler, and not a regular process.

A process, on the other hand, does not truly idle in kernel terms. When a process is not using the CPU (because it is waiting for I/O, a timer, or another event), it is in a *sleeping* or *blocked* state. Although it may appear inactive, it is still managed by the scheduler and may resume execution when its blocking condition is resolved.

The key distinction is that **only CPUs idle** in the kernel's formal sense. A CPU idles when it has no work to do, while a process never truly idles: it either runs, waits, or is terminated.

Chapter 4

Approaches and Tools for Container Energy Measurement

4.1 Introduction

Accurately measuring and attributing energy consumption in containerized environments has become a central challenge in sustainable cloud computing. As container orchestration platforms like Kubernetes grow in adoption, the need for energy observability at finer granularities (down to the container or even process level) becomes increasingly critical. This requirement stems from a range of applications, including cost optimization, carbon accounting, energy-aware scheduling, and performance tuning.

A number of tools and frameworks have emerged in recent years to address this problem. Some focus on the system or server level, exposing power metrics via standardized interfaces or external instrumentation. Others adopt telemetry-based estimation approaches that infer energy usage from resource utilization statistics. More recently, several tools have begun to target container-level energy attribution specifically, often by integrating with Kubernetes and leveraging technologies such as eBPF, RAPL, and cgroups.

This chapter surveys the landscape of existing tools, organized into three categories: system-level monitoring solutions, telemetry-based estimation frameworks, and container-focused energy attribution tools. Particular attention is given to tools that support Kubernetes environments, as these are directly relevant to the goals of this thesis. Each tool is analyzed based on its architecture, metric sources, modeling approach, and practical limitations. Later sections discuss the emerging tool KubeWatt and present a comparative synthesis of strengths and weaknesses across the reviewed solutions.

4.2 Non-container-focused Energy Monitoring Tools

4.2.1 Server-Level Energy Monitoring

While not directly translatable to container-level energy monitoring, server-level energy consumption remains an important aspect. Scientific works and tools in this domain generally do not provide the temporal resolution required for container-level energy monitoring.

4.2.1.1 Kavanagh and Djemame: Energy Modeling via IPMI and RAPL Calibration

Overview and Architecture Kavanagh and Djemame[12] present their findings on combining IPMI and RAPL (interface unspecified) data to estimate server energy consumption, achieving improved accuracy through calibration with an external server-level watt meter. For calibration, they induce artificial CPU workloads and rely on CPU utilization metrics with 1-minute averaging windows, necessitating extended calibration intervals to obtain stable readings. While the resulting model is tailored to their specific hardware and not generally portable, their work provides valuable insights into the complementary use of IPMI and RAPL. The authors recognize that the respective limitations of these tools (RAPL's partial scope and IPMI's low resolution) can be mitigated when used in combination.

Attribution Method and Scope Although the model operates at the physical host level, it supports attribution to VMs or applications using CPU-utilization-based proportional allocation. Several allocation rules are proposed, including utilization ratio, adjusted idle sharing, and equal distribution. However, no container-level attribution is attempted, and runtime flexibility is limited due to the static nature of the calibration.

Validation and Limitations With their watt-meter-calibrated model using segmented linear regression, the authors report an average error of just -0.17%. More relevant to practical application, they also construct a model based solely on IPMI and RAPL (calibrated via watt meter data), which achieves a reduced error of -5.58%, compared to -15.75% without calibration. Limitations of their approach include the need for controlled, synthetic workloads, coarse-grained sensor input, and the assumption of relatively stable system conditions during calibration.

Key Contributions

- **Hybrid use of IPMI and RAPL is analyzed**, showing that these tools compensate for each other's limitations. RAPL underestimates total system power, while IPMI captures more components but at lower resolution.
- IPMI accuracy is significantly improved through external watt meter calibration.
- The authors provide practical calibration guidelines:
  - Use long, static workload plateaus to align with averaging windows and reduce synchronization complexity.
  - Discard initial and final measurement intervals to avoid transient noise and averaging artifacts.
  - Ensure calibration workloads exceed the IPMI averaging window to capture valid steady-state values.

Relevance to Proposed Architecture This work informs the proposed architecture by demonstrating how combining RAPL and IPMI can yield more accurate system-level power estimation. The use of plateau-based calibration and composite data

models is especially applicable. However, the lack of container-level granularity, reliance on offline calibration, and limited attribution scope underscore the need for more dynamic, fine-grained, and container-aware approaches in Kubernetes-based environments.

4.2.1.2 CodeCarbon

CodeCarbon[107] is a Python package designed to estimate the carbon emissions of a program's execution. While its implementation is general-purpose, it is primarily aimed at machine learning workloads.

Overview and Architecture CodeCarbon estimates a workload's energy consumption by relying on RAPL *package-domain* CPU metrics via the `powercap` RAPL file system interface. A fix for the RAPL MSR overflow issue was implemented[108]. In the absence of RAPL support, it falls back to a simplified model based on the CPU's Thermal Design Power (TDP), obtained from an internal database, and combines it with CPU load metrics from `psutil`. For memory, a static power value is assumed based on the number and capacity of installed DIMMs. GPU power consumption is estimated via NVIDIA's NVML interface. The default measurement interval is 15 seconds, with the authors citing lightweight design as the primary motivation.

The component-level estimations are then aggregated and multiplied by a region-specific net carbon intensity (based on the local electricity grid's energy mix) to estimate the program's total CO_2 emissions. CodeCarbon is typically executed as a wrapper around code blocks, scripts, or Python processes.

Limitations There is no direct attribution of CPU activity to individual power metrics: CodeCarbon estimates energy use indirectly, based on the number of active cores and average CPU utilization, while making many assumptions that could be prevented. Combined with the relatively long measurement intervals, this results in background system processes also being attributed to the measured Python program. Consequently, CodeCarbon does not contribute directly to the goals of this thesis, which seeks fine-grained, container-level attribution.

However, the tool highlights several interesting secondary considerations. The integration of regional CO_2 intensity data is a valuable extension to conventional energy measurement and is well implemented. Additionally, the Python-based design offers high accessibility and ease of use, which may serve as inspiration for future developer-facing tools.

4.2.1.3 AI Power Meter

AI Power Meter[109] is a lightweight Python-based tool designed to monitor the energy consumption of machine learning workloads. It gathers power consumption data for the CPU and RAM via Intel RAPL using the `powercap` interface, and for the GPU via NVIDIA's NVML library. While the authors acknowledge that other system components (e.g. storage, network) also contribute to energy usage, these are not currently included and are considered an accepted limitation of the tool.

Unlike more advanced attribution tools, AI Power Meter does not distinguish between individual processes or workloads. Instead, it provides coarse-grained, system-level energy consumption measurements over time. In this respect, its scope is similar to *CodeCarbon*, focusing on ease of use and integration into ML pipelines rather than precise, per-process energy attribution. As such, while not directly applicable to container-level measurement or power attribution, AI Power Meter demonstrates the growing interest in accessible energy monitoring tools within the machine learning community.

4.2.2 Telemetry-Based Estimation Frameworks

4.2.2.1 PowerAPI Ecosystem[110] (PowerAPI, HWPC, SmartWatts)

PowerAPI[111] is an open-source middleware toolkit for assembling software-defined power meters that estimate real-time power consumption of software workloads. Developed as a generalized and modular framework, PowerAPI evolved alongside specific implementations such as *SmartWatts*, detailed in § 4.3.3. It allows power attribution at multiple granularity levels, including processes, threads, containers, and virtual machines. A distinctive strength of PowerAPI is the continuous self-calibration of its power models, enabling accurate real-time energy estimation under varying workloads and execution conditions. This makes PowerAPI particularly suited to heterogeneous computing infrastructures.

Overview and Architecture PowerAPI uses an actor-based model for modularity, enabling easy customization of its internal components with minimal coupling. It supports raw metric acquisition from diverse sensors (e.g. physical meters, processor interfaces, hardware counters, OS counters) and delivers power consumption data through various output channels (including files, network sockets, web interfaces, and visualization tools). As middleware, PowerAPI facilitates assembling power meters "*à la carte*" to accommodate specific user requirements and deployment scenarios.

Core Components

- **powerapi-core**: Middleware orchestrating real-time/post-mortem interactions between sensors and formulas. It defines the essential interfaces for sensor data ingestion and output channels (e.g. MongoDB, InfluxDB, CSV, socket, Prometheus), and includes built-in capabilities for data preprocessing, post-processing, and reporting.

- **hwpc-sensor**: A telemetry probe designed to gather low-level hardware performance counters (HWPCs), including instructions, cycles, and RAPL energy metrics. This sensor leverages *perf* and *cgroups-v2*, critical for fine-grained telemetry in containerized environments. It also provides detailed CPU performance state metrics via MSR events (`TSC`, `APERF`, `MPERF`).

- **SmartWatts-formula**[54]: A power model implementation (in Python) using HWPC data to estimate power consumption dynamically. It employs online linear regression provided by the Python *scikit-learn*[112] library, enabling accurate runtime learning of workload-specific power signatures. SmartWatts is further detailed in § 4.3.3.

- **SelfWatts-controller**: Dynamically selects hardware performance counters for software-defined power models, facilitating automatic configuration and unsupervised deployment in heterogeneous infrastructures. Currently, its development has stalled for several years, limiting its practical applicability.

- **pyRAPL**: A convenient Python wrapper around RAPL for CPU, DRAM, and iGPU energy metrics collection, providing easy access to hardware-based power data.

Relevance and Integration The modular and extensible architecture of PowerAPI positions it as a highly suitable foundation for further research and development of specialized power attribution tools. Researchers can readily extend or adapt its components to address evolving or niche requirements. However, its current implementation does not incorporate certain critical metrics, such as IPMI-based telemetry, which could limit its completeness in some practical deployment scenarios. Nonetheless, PowerAPI represents a significant advancement toward the creation of generalized, plug-and-play power models that operate without extensive manual calibration. This emphasis on practical deployability and general applicability highlights a key strength of the project and sets a clear direction for future research and development efforts in the domain of software-defined energy monitoring.

4.2.2.2 Green Metrics Tool

The *Green Metrics Tool* (GMT)[113] is an open-source framework designed to measure the energy consumption of containerized applications across various phases of the software lifecycle, including installation, boot, runtime, idle, and removal. It uses small, modular metric collectors to gather host-level energy and system data (e.g. CPU and DRAM energy via RAPL, IPMI power readings), and is orchestrated through declarative usage scenarios.

While GMT provides reproducible, lifecycle-aware measurements in controlled environments, it does *not* perform container-level or process-level energy attribution. The developers explicitly avoid splitting energy consumption across containers, citing the lack of reliable attribution models.

4.3 Container-Focused Energy Attribution Tools

While system-level monitoring and telemetry-based estimation provide valuable insights into overall server energy consumption, they fall short when it comes to attributing energy use to individual containers. In multi-tenant or microservice-based environments, such granularity is essential for accurate accountability, optimization, and scheduling decisions.

This section focuses on tools specifically designed to address this challenge by providing energy attribution at the level of containers or processes within a containerized environment. These tools typically integrate with container runtimes and Kubernetes, leveraging sources such as hardware counters, control groups, and performance monitoring frameworks to estimate or infer energy consumption.

The following subsections analyze three prominent tools (Kepler, Scaphandre, and SmartWatts), each with a distinct architectural approach. A fourth tool, KubeWatt, is discussed separately as a derivative implementation developed in response to identified limitations in Kepler.

4.3.1 Kepler

4.3.1.1 Overview and Goals

Kepler (*Kubernetes-based Efficient Power Level Exporter*)[49] is a modular, Kubernetes-native framework for monitoring, modeling, and estimating energy consumption in containerized environments. As the most prominent tool for container-level power estimation in Kubernetes, Kepler enables detailed observability of energy usage at the level of individual processes, containers, pods, and nodes[114].

Kepler integrates seamlessly with Kubernetes and Prometheus-based observability stacks. It supports both real-time energy metrics (e.g. RAPL, ACPI, NVML) and model-based estimation through trained regression models, making it applicable across a wide range of deployment environments, from bare-metal servers to virtual machines. Developed as an open-source CNCF project, Kepler's architecture is designed to be extensible, allowing researchers and practitioners to contribute new power models and adapt it to diverse system architectures.

It should be noted that shortly before the completion of this thesis, version 0.10.0 of Kepler was released. This version constitutes a major architectural rewrite of the project, intended to address structural limitations of earlier versions. However, the analysis in this chapter focuses on Kepler versions 0.9.x and earlier, which remain the most widely deployed at the time of writing. § 4.3.1.8 briefly summarizes the key changes introduced in the new release.

4.3.1.2 Architecture and Metric Sources

Kepler's architecture consists of several interconnected components, with the core functionality centered around a privileged monitoring agent that runs on every node. While the framework supports model-based estimation for environments without hardware telemetry, this thesis focuses on the direct collection of real-time power and utilization metrics available in bare-metal deployments.

Deployment Models Kepler supports multiple deployment scenarios depending on the availability of energy sensors on the host system. In bare-metal environments, Kepler can directly collect power metrics using RAPL, ACPI, or Redfish/IPMI interfaces. This is the most accurate and relevant mode for the purpose of this thesis. In contrast, on virtual machines (VMs), where access to hardware counters or power interfaces is restricted, Kepler relies on trained regression models to estimate node-level energy consumption. A third, currently unimplemented deployment model proposes a passthrough mechanism where a host-level Kepler instance would expose power metrics to a nested Kepler instance inside the VM. These deployment models are visualized in figure 4.1.

Kepler Agent and Exporter The core monitoring functionality is handled by the Kepler Agent, which is deployed as a privileged DaemonSet pod on each Kubernetes node. It collects energy and resource utilization metrics using a combination

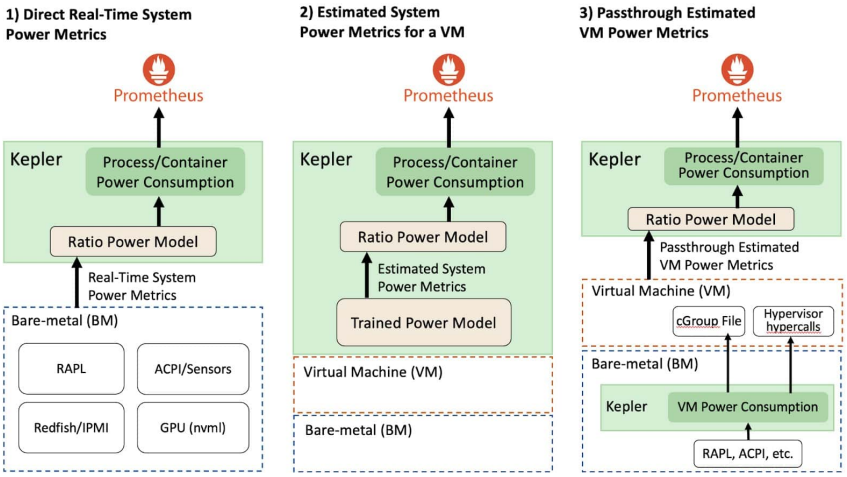


FIGURE 4.1: Kepler deployment models: direct power measurement on bare-metal, estimation on VMs, and the proposed passthrough model (currently not implemented)[115]

of eBPF instrumentation and hardware performance counters exposed via `perf_event_open`. A kprobe attached to the `finish_task_switch` kernel function enables accurate tracking of per-process context-switch activity. Container and pod attribution is performed after parsing the cgroup path from `/proc/<pid>/cgroup` and querying the Kubelet API for container metadata. The generated metrics are exported via a Prometheus-compatible endpoint for downstream processing and visualization. A generalized information flow is shown in figure 4.2.

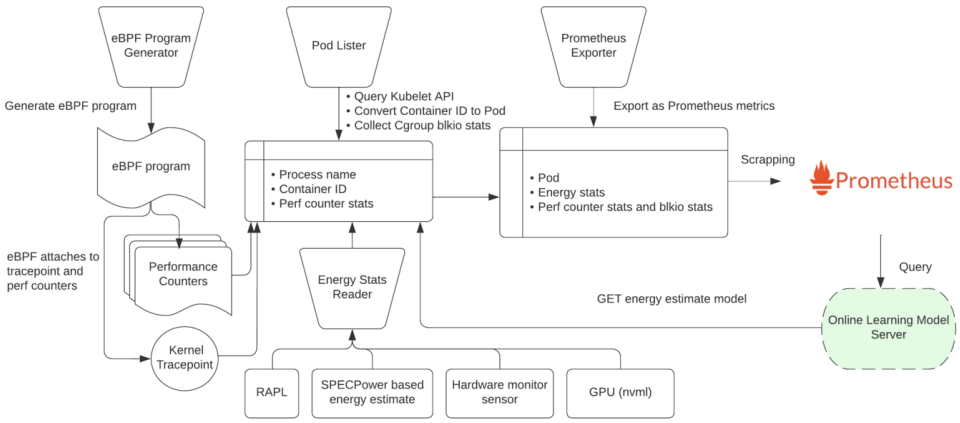


FIGURE 4.2: Simplified architecture of the Kepler monitoring agent and exporter components[115]

Resource Utilization via eBPF-based Hardware/Software Counters To measure low-level CPU activity, Kepler uses the Linux syscall `perf_event_open` to configure hardware performance counters on each core. The following events are tracked:

- `PERF_COUNT_HW_CPU_CYCLES`: Total CPU cycles (affected by DVFS)
- `PERF_COUNT_HW_REF_CPU_CYCLES`: Frequency-independent cycles
- `PERF_COUNT_HW_INSTRUCTIONS`: Retired instructions

- `PERF_COUNT_HW_CACHE_MISSES`: Last-level cache misses

These counters are accessed via BPF perf event arrays. On each context switch, current counter values are sampled, and deltas are computed against previously stored values. These deltas represent the CPU activity of the process leaving the CPU and are stored in BPF maps for later aggregation.

In addition to hardware counters, Kepler collects several software-level metrics that are not natively exposed by the Linux kernel. These include CPU time, page cache activity, and interrupt handling statistics. Because these metrics are unavailable through standard interfaces, Kepler uses custom eBPF programs to infer them from kernel behavior.

Since version 0.7, Kepler has migrated to `libbpf` and uses a BTF-enabled kprobe to instrument the `sched_switch` function. This allows Kepler to safely extract process IDs and timing data without relying on fragile symbol offsets. On each context switch, Kepler records timestamps and uses them to increment the `CPUTime` counter, providing fine-grained accounting of CPU residency per process.

Other software counters include:

- `PageCacheHit`: Tracks read and write access to the page cache using eBPF programs attached to `mark_page_accessed` and `writeback_dirty_folio`.
- `IRQNetTX`, `IRQNetRX`, `IRQBlock`: Count the number of softirq events attributed to a process, using the `softirq_entry` tracepoint.

Each of these metrics is manually accumulated in eBPF maps keyed by process ID and periodically read by the user-space collector. This enhances energy attribution, especially in scenarios where hardware counters are insufficient or unavailable.

Node Component-level Energy Consumption via RAPL Kepler supports component-level power estimation by reading RAPL energy counters, focusing on the `core`, `uncore`, `package`, and `dram` domains. The energy values are read via the `PowerCap` framework using the `/sys/class/powercap` interface. The sysfs path tree is parsed dynamically to detect available domains and sockets, ensuring compatibility across architectures and CPU generations. Energy values are read directly from files such as `energy_uj` and divided by 1000 to yield millijoule-level readings. A wraparound detection mechanism ensures robustness even when energy counters overflow.

The core logic is implemented in `UpdateProcessEnergy()`, which is invoked periodically by the main metrics collection loop (and also calls the process attribution logic immediately after the metrics update). However, despite RAPL's native ability to provide energy readings at approximately millisecond-level resolution, Kepler limits energy sampling to a coarse default interval of three seconds (defined in `config.SamplePeriodSec`). This choice reflects a trade-off between performance overhead and metric granularity but may limit accuracy for short-lived or bursty workloads.

Importantly, Kepler does not rely on eBPF or perf events to retrieve energy values; energy is obtained entirely through file-based reads from sysfs or, on some platforms, via MSR or hwmon fallbacks. The collected energy values are later exposed

to Prometheus and used in model training and runtime inference. The measurement cadence, attribution methodology, and available domains are validated using an internal tool that checks domain availability and collects average power readings across repeated samples.

Platform-Level Energy Consumption Kepler supports platform-level energy monitoring through external power interfaces exposed by the underlying server hardware. These measurements represent the total energy consumed by the entire node, as opposed to specific hardware components or processes. The implementation is modular, with each power source encapsulated in a corresponding `source` module. Currently supported backends include ACPI (via the `/sys/class/hwmon` interface), Redfish (via the Redfish REST API), and a stub for IBM's HMC interface on `s390x` systems.

Among these, Redfish provides the most detailed and reliable node-level power data. It queries the server's BMC for the `PowerConsumedWatts` value using a REST endpoint. This value is then converted into energy (in millijoules) by multiplying with the time elapsed since the previous query. Kepler spawns a background goroutine that polls this value at regular intervals (user-configurable via the `REDFISH_PROBE_INTERVAL_IN_SECONDS` parameter in the Kepler configuration). This design allows Redfish to provide cumulative energy measurements with known sampling resolution.

ACPI-based sources offer an alternative when Redfish is unavailable. These rely on instantaneous power averages and do not necessarily represent total node power. The HMC source, by contrast, is currently a non-functional placeholder used only on unsupported platforms. Overall, platform-level metrics are treated as node-wide aggregate energy values without internal attribution, but they offer valuable ground truth for cross-validating other metrics or monitoring infrastructure-level power trends.

Metadata Inputs for Container, System, and VM Attribution To organize energy and resource metrics by container, Kepler collects metadata that maps processes to Kubernetes pods, containers, and namespaces. Kepler supports both cgroups v1 and v2 and dynamically traverses `/sys/fs/cgroup` to map `PID` or `cgroupID` values to container IDs, extracting the last 64 characters of valid cgroup paths.

If the container ID is not cached, Kepler queries Kubernetes for pod metadata using either the Kubelet's local `/pods` API or, if enabled, the Kubernetes API server via a dedicated watcher. Both approaches extract metadata from `ContainerStatuses` fields in pod objects and populate a shared cache. This includes support for init and ephemeral containers. Container ID prefixes are stripped using regex to standardize the format.

Processes not associated with a container are labeled using fallback logic. If the process belongs to the root cgroup (`cgroupID = 1`) and cgroup-based resolution is enabled, it is labeled `container="kernel_processes"`, `namespace="kernel"`. All other unmapped processes are labeled `container="system_processes"`, `namespace="system"`. This includes host services such as `kubelet` and `containerd`. The Linux idle thread (PID 0), which does not appear in `/proc` and cannot be queried like regular processes, is not explicitly handled by Kepler; however, since

it belongs to the root cgroup implicitly, it is effectively included under the `kernel_processes` label.

In addition to container and system process tracking, Kepler supports experimental attribution for virtual machines running under QEMU/KVM on the host. When executed on the hypervisor, Kepler scans `/proc/<pid>/cgroup` for scope names matching the systemd pattern `machine-qemu-*.scope` to identify VM processes. If a match is found, the VM ID is extracted and used to create a `VMStats` structure, allowing the VM to be tracked similarly to a container. Optionally, a metadata lookup via the libvirt API can be used to resolve human-readable VM identifiers. This enables Kepler to expose VM-level resource and energy metrics on systems that mix containers and virtual machines. However, this mechanism only applies to Kepler instances running on the VM host and does not provide visibility into containers running inside the VM.

This metadata layer allows Kepler to cleanly separate containerized workloads, system processes, kernel activity, and virtual machines, ensuring complete coverage of energy attribution in subsequent stages.

GPU Power and Resource Utilization via NVML Kepler collects per-process GPU utilization statistics using the NVIDIA Management Library, accessed through internal Go bindings. Specifically, Kepler queries both compute engine usage and memory utilization for each process interacting with an NVIDIA GPU. This data is retrieved via the `ProcessResourceUtilizationPerDevice()` method, which internally calls NVML functions like `nvmlDevice.GetProcessUtilization` to return process statistics, including Streaming Multiprocessor utilization (`SmUtil`), memory utilization (`MemUtil`), and optional encoding/decoding activity. Additionally, Kepler collects GPU energy consumption using NVML's `getPowerUsage`.

To support NVIDIA's Multi-Instance GPU (MIG) architecture, Kepler first inspects whether MIG slices exist on a given device. If so, it iterates over each slice and retrieves per-process utilization data individually. Otherwise, it queries the full physical GPU. In both cases, the parent GPU ID is used as the key to unify resource attribution. For each PID returned by NVML, GPU utilization metrics are appended to the `ResourceUsage` field of the Kepler-internal `ProcessStats` structure. The following metrics are collected:

- `GPUComputeUtilization`: The percentage of compute engine usage (SM activity) over the sampling interval.
- `GPUMemUtilization`: The percentage of frame buffer memory usage per process.

Because NVML does not provide high-resolution energy counters, Kepler approximates GPU energy consumption by multiplying the instantaneous device-level power draw (`GetPowerUsage`) with the sampling interval duration (`SamplePeriodSec`). That is, energy per device is estimated as `energy=power(mW)` $\times$ `SamplePeriodSec`. This coarse-grained sampling, typically performed every few seconds, limits the temporal resolution and may miss short-lived GPU activity.

If `GetProcessUtilization()` is not supported by the hardware or fails at runtime, Kepler falls back to using `GetComputeRunningProcesses()` combined with

per-process memory usage to estimate GPU utilization. In this mode, energy is attributed proportionally to memory usage rather than compute activity. Alternative backends such as Habana or DCGM are supported but do not offer per-process utilization data and are thus not used for fine-grained attribution in Kepler's default configuration.

Summary of Inputs Tables 4.1 and 4.2 summarize the distinct types of input Kepler collects. Metric inputs provide raw resource and energy data, while metadata inputs allow this data to be mapped to containers, VMs, or system processes.

Metric Type	Source	Purpose
CPU hardware counters	`perf_event_open` via eBPF	Track cycles, instructions, cache misses per process
CPU software counters	Custom eBPF programs	Capture CPU time, IRQs, page cache activity
RAPL energy counters	`/sys/class/powercap`	Direct energy measurement for CPU and memory domains
Platform power	Redfish REST API, ACPI sysfs	Estimate total node energy from external sensors
GPU utilization	NVML (via Go bindings)	Track per-process compute and memory usage
GPU energy (approx.)	NVML power reading × interval	Estimate per-device energy usage

TABLE 4.1: Metric inputs used by Kepler for energy and resource monitoring

Metadata Type	Source	Purpose
Container ID extraction	`/proc/<pid>/cgroup`	Identify container context of each process
Pod and namespace info	Kubelet API or Kubernetes API server	Map container ID to pod, namespace, container name
System process fallback	Internal constants + missing container ID	Label processes outside containers as `system_processes`
Kernel thread fallback	`cgroupID = 1`	Label root-cgroup processes as `kernel_processes`
Virtual machine ID	`machine-qemu-*.scope` + optional libvirt lookup	Label QEMU/KVM-based VMs by scope or libvirt metadata
GPU process association	PID-based matching via NVML	Associate GPU usage with Linux processes

TABLE 4.2: Metadata inputs used by Kepler to organize and label monitored workloads

Export Interface and Metric Exposure All metrics collected by Kepler are ultimately exposed via Prometheus after power attribution. This includes both raw utilization metrics (e.g. CPU time, instructions, cache misses) and derived energy metrics (e.g. power estimates per container). The Prometheus export format allows flexible time series queries and integration with Grafana or other observability platforms.

Estimator Sidecar and Model Inference In addition to its lightweight internal estimator, Kepler optionally supports an estimator sidecar container that uses more sophisticated regression models for power inference. The sidecar communicates with the exporter via a Unix domain socket and loads pre-trained models (e.g. Scikit-learn, XGBoost) suitable for online estimation in environments lacking direct power metrics. This mechanism is mainly intended for telemetry-sparse environments and is not used in the real-time deployment mode considered in this thesis.

Model Server and Model Training For environments where energy measurement is available (e.g. via RAPL or Redfish), Kepler provides a separate model server that facilitates training of machine learning-based energy models. This component ingests time-aligned Prometheus metrics and energy readings, applies preprocessing steps, and generates regression models that estimate power consumption based on utilization metrics. Both absolute and dynamic models can be trained, using different feature groups (e.g. time-based vs. instruction-based) and labeled according to power domain (e.g. core, package). The resulting models are stored in a hierarchical directory format and can be deployed in other environments via the estimator sidecar. The model server supports modular pipelines, allowing flexible input data sources, energy isolation techniques, and regression backends. This training loop is essential in contexts where hardware telemetry is available during development or profiling but not in production. While model training plays a key role in extending Kepler to new environments[116], it is not central to the goals of this thesis.

External Integration Kepler is designed to integrate seamlessly with existing observability tools. Prometheus is used to scrape metrics from the Kepler agent, and dashboards can be built using Grafana to visualize energy consumption across nodes, containers, or applications. While not required for core functionality, this integration facilitates operational awareness and debugging.

4.3.1.3 Attribution Model and Output

Kepler Configurability and Measurement Interval Kepler updates its input metrics at a configurable interval (default: `SamplePeriodSec` = 3 seconds), with the exception of Redfish-based measurements, which have a separate, user-defined interval (default: 15 seconds). Although utilization and RAPL energy metrics theoretically support more fine-grained sampling, Kepler performs attribution solely based on aggregated values within each configured sampling window. While this reduces system overhead, it may miss short-lived energy fluctuations. Kepler's attribution framework is modular by design, allowing easy integration of new power sources through standardized interfaces. Depending on the availability of input metrics, Kepler uses either direct power readings or model-based estimation. In the absence of hardware telemetry (i.e., RAPL), Kepler can apply regression models trained on a reference system where accurate power readings are available. For this thesis, only scenarios where all relevant metrics are available (as discussed in § 4.3.1.2) are considered.

Division of Power into Idle and Dynamic Components Kepler conceptually divides total power consumption into *idle power* and *dynamic power*. Their sum constitutes the total (often named *absolute*) measured energy on both process and node levels. Dynamic power is correlated with resource utilization, while idle power represents the static baseline consumption that persists regardless of system activity.

This distinction is critical, as Kepler distributes these two components differently among processes [117]. Idle power attribution follows the Greenhouse Gas Protocol guidelines [118], which recommend splitting idle power across containers based on their size (i.e., resource requests relative to the total). Dynamic power, by contrast, is attributed proportionally to observed resource usage.

Node-Level Energy Attribution On the node level, Kepler directly uses RAPL energy readings from the `pkg`, `core`, `uncore`, and `dram` domains for all available sockets, reporting energy in millijoules. GPU energy is measured per device via NVML. Platform-level energy is retrieved via the configured source, typically Redfish. Kepler iterates through all exposed chassis members and internally converts Redfish power metrics (in watts) to milliwatts, which are then multiplied by `SamplePeriodSec` to produce energy values in millijoules. These can then be treated like standard metrics.
After collecting node-level energy metrics, Kepler updates the tracked *minimum idle power* if a new lower value is observed. The corresponding resource usage is also recorded. This tracking applies to all RAPL domains and GPU devices. Node-level dynamic energy is computed as the difference between total and idle energy. Additionally, Kepler calculates the residual category *Other* as platform power minus the sum of `pkg`, `dram`, and GPU power, separately for idle and dynamic metrics. This implicitly attributes all remaining energy to unspecified components (e.g. network interfaces, fans, or storage devices).

Process-Level Energy Attribution Process energy is attributed based on resource utilization metrics and node-level energy values. Using the most recent measurements of component, GPU (if enabled), and platform power, Kepler calculates each process's energy consumption. Idle energy is simply divided evenly among all processes. This is in contrast to Kepler's documentation, which states that idle energy should be divided by requested container size (in accordance with the GHG protocol [118]). Since container-level metadata is aggregated from per-process statistics, implementing size-based idle attribution at the process level is non-trivial. Nonetheless, this feature is advertised but not implemented, as noted by a `TODO` comment in the codebase.

Dynamic energy is attributed proportionally using the formula:

$$E_{process_{dyn}} = \left\lceil E_{component_{dyn}} \cdot \frac{U_{process}}{U_{node}} \right\rceil \tag{4.1}$$

where U_{process} is the process's resource usage, and U_{node} is the total component resource usage indicated by RAPL. If no usage is recorded, energy is divided evenly among processes. The specific usage metrics used in the default ratio model are:

- **CPU**: based on `CPUInstructions` (fallback: `CPUTime`)
- **DRAM**: based on `CacheMisses` (fallback: `CPUTime`)
- **GPU**: based on `GPUComputeUtilization` (via NVML)
- **Other** and **Platform**: intended to use a configurable default metric

However, the default usage metric for `Uncore` and `Platform` domains is not set in the configuration, defaulting to an empty string. As a result, these metrics are not attributed based on usage and fall back to equal distribution. It is unclear whether this is a conscious design decision or an oversight.

The following formula 4.2 summarizes the complete energy attribution model used by Kepler at the process level. It combines all relevant energy domains: dynamic and idle energy across measured components (e.g. package, core, DRAM, uncore, GPU), as well as residual platform energy not accounted for by specific components. Each part of the equation reflects a distinct step in the attribution logic previously discussed, merging them into a single expression for total per-process energy consumption.

$$
\begin{aligned}
E_{\text{process}} = & \sum_{\text{comp} \in \{\text{core,dram,uncore,gpu}\}} \left(\left\lceil E^{\text{dyn}}_{\text{comp}} \cdot \frac{U_{\text{process, comp}}}{U_{\text{node, comp}}} \right\rceil + \left\lceil \frac{E^{\text{idle}}_{\text{comp}}}{N_{\text{processes}}} \right\rceil \right) \\
& + \left\lceil \frac{E^{\text{dyn}}_{\text{platform}} - \left(E^{\text{dyn}}_{\text{pkg}} + E^{\text{dyn}}_{\text{dram}} \right)}{N_{\text{processes}}} \right\rceil + \left\lceil \frac{E^{\text{idle}}_{\text{platform}} - \left(E^{\text{idle}}_{\text{pkg}} + E^{\text{idle}}_{\text{dram}} \right)}{N_{\text{processes}}} \right\rceil
\end{aligned} \tag{4.2}
$$

4.3.1.4 Attribution Timing and Export Granularity

Kepler attributes energy at fixed internal intervals (default: 3 seconds). These deltas are exposed via Prometheus metrics such as `kepler_container_joules_total`, but the Prometheus scrape interval is user-defined and may not align with Kepler's update loop. If the export interval is not a multiple of the internal interval (e.g. 5s vs. 3s), metric misalignment can occur, leading to visible fluctuations even for stable workloads. Since Kepler does not smooth or interpolate values, this behavior is expected and intentional. For more stable time series, it is recommended to configure the export interval as a multiple of Kepler's internal interval.

This timing issue primarily affects metrics that are reported as cumulative deltas (e.g. energy in millijoules). For Redfish-based metrics, which are provided in watts and converted internally by Kepler into energy values by multiplying with the update interval, synchronization is less problematic. The conversion to energy is performed over a well-defined window, which inherently matches the Prometheus export interval, thus avoiding the same class of timing issues.

4.3.1.5 Validation and Research Context

Kepler has been adopted in various academic and industrial settings as a tool for estimating energy consumption at the container and pod level. In most cases, it is treated as a black-box exporter, with little investigation into the reliability or internal consistency of its reported metrics. While its integration with Kubernetes and Prometheus makes it convenient to use, its internal attribution mechanisms and modeling assumptions are rarely scrutinized in published work. Andringa[119] investigates Kepler, but lacks the necessary depth.

A notable exception is the study by Pijnacker et al. [50, 52], which constitutes the first dedicated empirical evaluation of Kepler's behavior and attribution accuracy.

Their work critically examines whether Kepler's per-container energy metrics correspond to expected utilization and measured system-level power under controlled conditions.

To this end, the authors design a testbed using a Dell PowerEdge R640 with dual Intel Xeon CPUs, operating a single-node Kubernetes cluster. Redfish/iDRAC power readings, validated using a wall power plug, are used as a high-confidence ground truth reference. Container CPU metrics and metadata are collected using Prometheus and cAdvisor. Kepler is deployed in version 0.7.2, configured to use RAPL for component-level power metrics and Redfish for platform-level readings.

The validation experiment employs a repeated CPU stress workload (`stress-ng -cpu 32`), combined with a set of idle containers left in the Completed state to test the attribution of idle power. Additional tests involve dynamic workload changes, including the deletion of idle pods during active load, to analyze Kepler's behavior during transitions in cluster state. These experiments reveal attribution inconsistencies across both idle and dynamic workloads.

Importantly, the authors isolate the source of Kepler's inaccuracy as its attribution logic, rather than its raw power estimation. While node-level energy estimates can match ground-truth readings closely, the logic used to assign portions of that energy to specific containers often fails to reflect actual utilization patterns. Inconsistent handling of kernel and system processes, as well as temporary spikes in attribution caused by metric timing mismatches, point to deeper architectural shortcomings. These and other limitations are summarized in the next subsection.

4.3.1.6 Limitations and Open Issues

The results of the evaluation by Pijnacker et al. reveal several systemic issues in Kepler's container-level energy attribution. In parallel, this thesis identifies additional design and implementation problems through source code analysis and direct experimentation.

Validation results from Pijnacker et al. [52]:

- **Incorrect idle power attribution:** Kepler assigns idle power equally to all containers, including pods in the Completed state. This leads to energy being attributed to containers that are no longer active, skewing total container-level metrics.
- **Latency mismatch artifacts:** A mismatch between fast-updating utilization metrics (e.g. CPU usage) and slower power metrics (e.g. Redfish at 60s intervals) causes transient attribution spikes. This effect is most pronounced during workload transitions.
- **Inconsistent attribution to system processes:** When idle containers are removed or when attribution is unclear, Kepler sometimes redirects energy consumption to generic "system processes", even when this does not match observed CPU activity. This fallback mechanism introduces further attribution noise.

- **Unstable behavior during dynamic cluster state changes:** In deletion experiments, the removal of idle pods triggered inconsistent redistribution of both idle and dynamic power, including unexplained reductions in power attributed to active workloads.

- **Observability gaps for Completed pods:** Once a pod transitions to the Completed state, it may no longer be visible to cAdvisor or other telemetry tools. This can lead to orphaned power assignments in Kepler, which continues to attribute energy to containers that are no longer observable.

Additional issues identified in this work While the attribution issues demonstrated by Pijnacker et al. are the most critical and empirically validated shortcomings of Kepler, a detailed inspection of the source code conducted as part of this thesis (see § 4.3.1.3) confirms these problems and reveals several additional implementation and documentation-related concerns. Although these issues are less severe in their impact, they further highlight the current limitations of Kepler as a mature and reliable observability tool.

- **Incomplete codebase:** Kepler's source code contains over 1000 unresolved `TODO` markers, some located in critical modules such as the ratio-based power model. These incomplete sections may affect reliability or create hard-to-debug behavior in real deployments.

- **Discrepancy between documentation and implementation:** While Kepler's documentation describes proportional idle power distribution based on container size or usage, the actual implementation uses equal distribution across all containers, regardless of their activity or footprint.

- **Outdated documentation and diagrams:** Both public-facing and internal documentation, including UML diagrams and developer notes, are out of date. This lack of alignment between the codebase and its documentation hinders reproducibility and extension.

- **Inadequate measurement intervals for heterogeneous workloads:** Although Kepler collects advanced eBPF-based metrics at high frequency, RAPL-based measurements are only sampled every three seconds. For workloads with heterogeneous container behavior or short-lived processes, this can drastically impact accuracy. Higher sampling intervals for RAPL-based data would allow more fine-grained attribution.

Despite these limitations, it is important to emphasize that **Kepler remains the most advanced and integrated open-source implementation for process-level energy monitoring in Kubernetes environments to date**. Its combination of kernel-level instrumentation, Prometheus integration, and extensible modeling capabilities makes it a uniquely valuable reference point and starting foundation for future research and tool development.

4.3.1.7 KubeWatt (Derived from Kepler)

As a direct response to the shortcomings identified in Kepler's attribution model, Pijnacker developed *KubeWatt*, a proof-of-concept alternative designed for improved container-level energy observability in Kubernetes. While Kepler's modular design

and eBPF-based telemetry collection provide flexibility, it suffers from attribution artifacts, particularly related to static power division, temporal misalignments, and system process handling. KubeWatt explicitly addresses these issues by adopting a simpler, more transparent approach centered around external power readings and CPU-based attribution.

Separation of Static and Dynamic Power A core improvement in KubeWatt is its strict separation between static and dynamic power. Unlike Kepler, which distributes idle power among all containers, including non-running ones, KubeWatt isolates static power entirely. Static power, including control plane activity and system overhead, is measured upfront using one of two initialization modes. It is then excluded from per-container attribution. Dynamic power is attributed solely to active containers based on their proportional CPU utilization. This separation eliminates attribution artifacts such as idle containers appearing to consume power.

Initialization Modes for Static Power Estimation KubeWatt introduces two initialization modes to measure or estimate static power: *base initialization* and *bootstrap initialization*. The base mode measures node power in an empty cluster to establish an accurate baseline, while the bootstrap mode estimates static power during active workloads by fitting a third-degree polynomial regression model to observed CPU and power values. The initialization result is stored and reused, based on the assumption that static power remains stable over time.

Simplified Attribution Model For container-level attribution, KubeWatt modifies the hierarchical power-mapping model originally proposed in [119]. It attributes dynamic node power to containers proportionally to their CPU usage, normalized over all active containers. (While the specific metric used for *CPU usage* is a central feature of Kepler, KubeWatt does address this specifically.) This avoids using the full node CPU metric, which includes kernel and system overhead already covered by static power. The attribution model is transparent, avoids container misclassification, and is robust against changes in container count.

Resilience Against Kepler's Pitfalls In contrast to Kepler, KubeWatt shows robustness under dynamic workload changes, such as container creation or deletion. It avoids misattributing power to terminated or inactive pods and mitigates power leakage into "system_processes" labels. Additionally, KubeWatt explicitly ignores control plane pods in the dynamic attribution, instead incorporating their idle load into the static baseline.

External Power Source Integration Unlike Kepler, which may rely on internal sensors or hybrid models, KubeWatt is built around a single, clearly defined power source: the Redfish API via iDRAC. Power data is abstracted via a modular interface, allowing extensibility to other external sources. This design choice avoids mixing sensor domains and contributes to more interpretable results.

Improved Evaluation Outcomes Empirical evaluation confirms that KubeWatt reduces RMSE in total node power estimation compared to Kepler, achieves accurate attribution even under stressor workloads, and avoids over-reporting container power. Notably, the estimator mode maintains high accuracy as long as container CPU utilization is stable and consistent with Prometheus scrape intervals.

Overall, KubeWatt represents a targeted refinement of Kepler's architectural and attribution approach, favoring interpretability and correctness over feature completeness. While limited in scope (e.g. no support for GPU or memory attribution), it successfully mitigates several of the critical limitations identified in Kepler.

4.3.1.8 Re-Release of Kepler Version 0.10.0

During the final stages of this work, Kepler version 0.10.0 was released as a significant architectural rewrite. While maintaining its fundamental role as a Prometheus exporter for container-level energy metrics, the new release introduces considerable changes to both its internal implementation and energy attribution methodology. This section provides an overview of the revised architecture and critically analyzes the implications of these changes in the context of container-level energy observability.

Implementation Changes Kepler v0.10.0 represents a ground-up redesign focusing on modularity, thread-safety, and accessibility. Mutex locks are used extensively throughout the codebase to ensure safe concurrent metric collection. Unlike previous versions, Kepler now requires only read-only access to `/proc` and `/sys` filesystems, eliminating the need for elevated privileges (`CAP_SYSADMIN`, `CAP_BPF`). This is marketed as a significant improvement in deployability and security.

Container detection still relies on traditional cgroup path analysis, with per-process cgroups parsed via regex matching to identify container runtimes and IDs. However, unlike prior versions, container-level resource metrics are no longer derived by aggregating per-process metrics. Instead, Kepler v0.10.0 treats containers as first-class workloads: CPU usage statistics for containers are read directly from cgroups, without requiring process-level aggregation.

Node-level CPU activity is determined via the Linux `/proc/stat` interface, using the formula:

$$\text{CPUActiveTime} = \text{TotalTime} - (\text{IdleTime} + \text{IOWaitTime}) \tag{4.3}$$

A node-level usage ratio is computed as:

$$\text{CPUUsageRatio} = \frac{\text{CPUActiveTime}}{\text{TotalTime}} \tag{4.4}$$

This represents a significantly simplified utilization model compared to prior versions, which relied on more granular hardware performance counters (e.g. instructions retired, cache misses). While the new architecture is once again designed in a modular way, this specific ratio model is not, perhaps suggesting that no user-selectable utilization metric is aimed for, as there was in the previous version.

At the power measurement layer, Kepler continues to rely on RAPL readings, now referred to as "energy zones". In addition to standard `package`, `core`, `uncore`, and `dram` zones, Kepler v0.10.0 dynamically detects the `psys` domain, expanding applicability to non-server-grade processors. Multi-socket systems are supported via automatic aggregation of identically named RAPL zones, with counter wraparounds

correctly handled through delta calculations based on maximum energy values.

Finally, pod lifecycle awareness has seen notable improvements. Kepler now tracks ephemeral and init containers explicitly, indexing container IDs via Kubernetes' native API server. This mitigates previous observability gaps for short-lived and Completed pods, which were previously problematic due to reliance on cAdvisor and cgroup detection alone.

Power Attribution Model The energy attribution model has undergone a substantial simplification. Kepler v0.10.0 attributes energy solely based on CPU time. The node's total RAPL-reported active energy is distributed proportionally to workload CPU activity:

- **Node-level attribution:** At each interval, total energy deltas are computed from RAPL readings. Active and idle energy components are calculated using the node's CPU usage ratio:

$$E_{\text{active}} = \Delta E \cdot \text{CPUUsageRatio} \tag{4.5}$$

$$E_{\text{idle}} = \Delta E - E_{\text{active}} \tag{4.6}$$

- **Process-, container-, and pod-level attribution:** Processes, containers, and pods are treated as independent workloads. Each receives a fraction of the node's active energy based on its CPU time share:

$$E_{\text{workload}} = E_{\text{node, active}} \cdot \frac{\text{CPUTimeDelta}_{\text{workload}}}{\text{CPUTimeDelta}_{\text{node}}} \tag{4.7}$$

It is important to emphasize that container metrics are no longer derived from per-process metrics. Instead, both processes and containers draw energy directly from node-level active energy based on their own independently tracked CPU usage statistics. This parallel accounting structure avoids process-level aggregation.

Critical Analysis While Kepler v0.10.0 improves accessibility, security, and operational simplicity, these enhancements come at the cost of reduced modeling fidelity. CPU time has replaced more nuanced metrics such as CPU instructions and cache misses for workload attribution. The new model assumes linear energy scaling with CPU activity, disregarding workload heterogeneity and hardware-level power behaviors such as frequency scaling or turbo boost. By not attributing idle power to workloads, Kepler focuses on actionable energy consumption linked to active resource use. While this limits full node-level energy accountability at workload granularity, it avoids inaccuracies introduced by arbitrary idle power distribution.

Additionally, Kepler's model assumes that CPU time directly correlates to power consumption, overlooking variations introduced by CPU frequency scaling, idle states, instruction-level differences, and workload-specific resource behaviors (e.g. memory-bound tasks). These factors can significantly distort power attribution, especially in heterogeneous or bursty workloads.

rom a research perspective, the simplification represents a methodological compromise: Kepler prioritizes operational deployability and stability over modeling granularity. While suitable for general monitoring, its current attribution model is less suited to detailed workload-specific energy analysis.

In the author's view, Kepler's redesign trades attribution accuracy for simplicity and usability. While this architectural shift benefits practical deployment, it diminishes the tool's value for research-focused energy observability. Finer-grained temporal attribution and more diverse utilization metrics would have been desirable developments.

4.3.2 Scaphandre

4.3.2.1 Overview and Goals

Scaphandre[120] is an energy monitoring agent designed to expose power consumption metrics at fine granularity, particularly in containerized and virtualized environments. Its name, derived from the French word for "diving suit", reflects its goal of providing deep insights into system-level energy consumption.

Scaphandre aims to attribute energy usage to individual processes, containers, or pods. It supports multiple deployment models: it can be installed directly on a physical host (where it can also monitor qemu/KVM-based virtual machines) or deployed in a container, measuring the energy usage of the host system, provided that the `/sys/class/powercap` and `/proc` directories are mounted as volumes. Most relevant to this thesis, Scaphandre can also be deployed on a Kubernetes cluster via a Helm chart, optionally alongside *Prometheus* and *Grafana*, allowing for convenient monitoring of Kubernetes nodes and containers.

4.3.2.2 Architecture and Metric Sources

Scaphandre is written in Rust and features a modular, extensible architecture built around two core components: *sensors* and *exporters*. Sensors gather energy-related data from the host system, while exporters format and expose this data to external systems. This design allows seamless integration into cloud-native observability stacks and automation pipelines.

Sensors Scaphandre's *Sensors* subsystem collects utilization and energy consumption metrics from the host system and makes them available to exporters. An abstraction layer, implemented in `sensors/mod.rs`, manages system topology (tracking sensors, CPU sockets, RAPL domains, and processes) and manages the generation of metric records.

The default Linux implementation, `PowercapRAPLSensor` (located in `powercap_rapl.rs`), constructs a power topology by reading Intel RAPL data from the `/sys/class/powercap` interface. By default, it identifies all available domains (and sockets) by matching directories with pattern `intel-rapl:<socket>:<domain>`, each containing a domain name and an `energy_uj` file reporting cumulative energy in microjoules. If no domain-level directories are found, Scaphandre triggers a fallback mechanism that omits subdomain detail and instead reads from the socket-level file, if available. This file represents the `package` (PKG) domain, which aggregates the energy consumption of the entire CPU socket. While PKG is a standalone

RAPL domain, it also serves as the root of the domain hierarchy. Notably, Scaphandre does not currently handle overflow in the `energy_uj` counter, a known limitation that has not yet been resolved despite user-reported inaccuracies [121].

In addition to the Powercap-based sensor, a Windows-only sensor is available that reads energy consumption directly from Intel or AMD-specific Model-Specific Registers (MSRs).

System utilization metrics are collected by helper functions implemented in `utils.rs`, which rely on Rust's `sysinfo` and `procfs` crates. These libraries extract data from virtual filesystems such as `/proc`, `/sys`, and `/dev`. When container support is enabled, cgroups are also read, but solely to map processes to containers. The collected metrics are stored in a custom data structure along with timestamps, while cgroup information is attached to each process as additional metadata labels. Apart from metrics of processes, CPU and memory, disk read/write operations are also collected.

The *sensors* abstraction layer also applies key data preprocessing steps, which are discussed in § 4.3.2.3.

Exporters *Exporters* are responsible for collecting metrics from sensors, storing them temporarily, and exporting them to external systems. Crucially, they also perform the attribution of energy and system metrics to specific scopes, which is discussed in detail in the following section. Similar to the sensor subsystem, Scaphandre uses an abstraction layer (implemented in `exporters/mod.rs`) to manage common exporter logic such as metric preparation and attribution. Individual exporters are implemented as pluggable modules. The `MetricGenerator` plays a central role by consolidating and attributing metrics across all scopes: the Scaphandre agent itself, the host, CPU sockets, RAPL domains, system-wide statistics, and individual processes.

The modular exporter architecture enables easy implementation of new output formats, and Scaphandre explicitly encourages developers to implement custom exporters if needed. Currently, exporters exist for *Prometheus*, *Stdout*, *JSON*, and *QEMU*, among others.

The *QEMU* exporter is particularly notable. It is designed for use on QEMU/KVM hypervisors and allows energy metrics to be exposed to virtual machines (VMs) in the same way that the `powercap` kernel module does on bare-metal systems. Specifically, the QEMU exporter writes VM-specific energy metrics to a virtual file system. Inside the VM, a second Scaphandre instance can read these files as if they were native energy interfaces. This mechanism mimics a bare-metal powercap environment, allowing software running in the VM to access energy data without direct access to RAPL or hardware sensors. This architecture is especially useful for breaking the opacity of power monitoring in virtualized environments, where such metrics are usually unavailable. If adopted by cloud providers, this could significantly improve visibility into energy consumption within public cloud VMs, supporting energy-aware software design even in virtualized infrastructures.

Measurement Interval Scaphandre's measurement interval is fixed at two seconds, as defined in the `show_metrics` function of the file `/exporters/prometheus.`

`rs`. This hardcoded interval determines how frequently the agent performs a new measurement by reading cumulative energy values from the RAPL interface. While RAPL counters are updated approximately every millisecond, Scaphandre does not take advantage of this high-resolution data. Reducing the interval could improve measurement accuracy and temporal granularity (especially for short-lived processes) but would also increase system overhead. As of now, modifying the interval would require changes to the source code.

4.3.2.3 Attribution Model

Scaphandre attributes power consumption to processes and containers using a proportional model based on CPU usage over time. At each sampling interval, it reads the system's total energy consumption from Intel RAPL counters and distributes this energy among all active processes according to their normalized CPU utilization.

The core of this attribution logic is based on CPU time as reported in `/proc`. For each process, Scaphandre accumulates CPU time in active states and excludes time spent in inactive states such as `idle`, `iowait`, `irq`, and `softirq`. This filtering is applied both at the per-process and system level, so that only time considered "active" is included in the normalization denominator. The result is a time-based utilization model that excludes idle-related CPU time from consideration.

Per-process energy is then computed using the following formula, conceptually implemented in `get_process_power_consumption_microwatts()`:

$$E_{\text{proc}}(t) = \frac{E_{\text{RAPL}}(t) \cdot \text{CPU}_{\text{proc}}(t)}{\sum_i \text{CPU}_i(t)} \tag{4.8}$$

where $E_{\text{RAPL}}(t)$ denotes the energy delta over the sampling interval t, and $\text{CPU}_{\text{proc}}(t)$ is the active CPU time of the process. The denominator includes the sum of active CPU time across all processes, excluding inactive states.

Container-level attribution is achieved by inspecting the cgroup path of each process and mapping it to a container or Kubernetes pod using runtime-specific metadata. When container context is available, Scaphandre enriches its per-process metrics with labels such as `container_id`, `kubernetes_pod_name`, and `namespace`. These metrics, such as `scaph_process_power_consumption_microwatts`, are then exposed via Prometheus and can be aggregated at container or pod level.

In addition to process- and container-level metrics, Scaphandre also reports host-level power consumption using the *PSYS* RAPL domains. These host metrics include total platform or package energy consumption, as available, and are exposed via metrics such as `scaph_host_power_microwatts` and `scaph_host_energy_microjoules`. Notably (as pointed out in § 2.4.2), the *PSYS* domain typically does not exist on server-grade systems, in which case Scaphandre adds the *PKG* and *DRAM*-packages to calculate host consumption. Unlike per-process energy values, these host-level metrics reflect total system energy use and are not adjusted to exclude idle CPU time.

Idle power consumption Scaphandre does not compute idle power directly, but it is possible to infer a residual estimate by comparing the total reported host power

to the sum of per-process power:

$$P_{\text{idle}} = P_{\text{Host}} - \sum P_{\text{proc}}$$

This difference may include contributions from idle power, system activity outside of user processes, or RAPL domain coverage gaps. However, this value is not part of Scaphandre's exported metrics and must be computed externally if needed.

This attribution methodology is used consistently across the Scaphandre exporters, including the Prometheus exporter. It forms the basis for container-aware energy attribution without requiring additional instrumentation or hardware performance counter support. Further discussion of this methodology's strengths, assumptions, and limitations follows in the next section.

4.3.2.4 Validation and Research Context

To date, no formal validation of Scaphandre's attribution methodology has been published. While the underlying RAPL interface used for energy measurement is widely accepted and validated in prior research, the specific proportional attribution model employed by Scaphandre has not been systematically evaluated against ground-truth data or instruction-based models. Despite this, Scaphandre has been used in academic and applied contexts for estimating the energy consumption of software systems. In most cases, it is treated as a black-box exporter of container-level energy metrics, with limited investigation into its internal measurement and attribution logic.

A more detailed assessment is presented by Tarara[104], who compares Scaphandre's reported per-process power consumption against both CPU utilization and instruction-based measurements. His case study reveals that Scaphandre tends to overestimate power consumption for lightweight processes under low-load conditions, due to its policy of distributing all observed energy among the small set of active processes. While the model excludes idle time from CPU usage calculations, it does not exclude idle power from total energy, leading to attribution artifacts. Tarara concludes that Scaphandre improves upon naïve utilization-based models but cannot match the precision of instruction-level approaches using tools such as `perf`.

4.3.2.5 Limitations and Open Issues

While Scaphandre offers a lightweight and transparent approach to energy attribution, several limitations emerge from its current implementation.

First, its fixed measurement interval of 2 seconds limits temporal resolution. Although the underlying RAPL interface supports much finer granularity, Scaphandre aggregates CPU activity over 2-second windows. During this time, the Linux scheduler may switch between dozens or hundreds of processes, depending on system activity. As a result, Scaphandre can only attribute energy based on average CPU utilization over the interval, potentially masking short-lived or bursty behavior.

Second, Scaphandre attempts to refine CPU-based attribution by subtracting inactive time (`idle`, `iowait`, `irq`, `softirq`) from the denominator of its proportional model. While this adjustment excludes clearly non-active states, it does not fully resolve the underlying ambiguity of CPU utilization as a proxy for energy. As noted by

Tarara[104], when overall system load is low, the total observed energy (especially idle platform power) is still fully distributed among a small set of active processes. This can result in inflated or misleading per-process energy values, particularly for lightweight workloads.

Scaphandre relies exclusively on Intel's RAPL interface for energy measurements. The developers of Scaphandre acknowledge the lack of detailed public documentation, especially for the `PSys` domain, which they assume to represent total SoC power. Their experiments also highlight ambiguity regarding domain overlaps (for instance, whether the `DRAM` domain is already included in `PKG`, or whether it must be treated as a separate component). These uncertainties may affect the interpretation of host-level energy metrics and the completeness of total system energy accounting.

In a comparison experiment between Scaphandre and other tools, Raffin[16] was unable to push Scaphandre's measurement frequency beyond 28 Hz, indicating a suboptimal implementation. Despite being written in Rust, Scaphandre performed significantly worse than the Python-based tool *CodeCarbon*. Raffin measured Scaphandre's overhead at a non-negligible 3-4% at 10 Hz on an Intel server.

Finally, Scaphandre does not incorporate instruction-level or performance counter-based metrics. It cannot distinguish between processes with different execution intensities or memory behavior and assumes a uniform relationship between CPU time and energy use. This limits its ability to reflect differences in instruction throughput, stalling, or architectural efficiency across workloads.

4.3.3 SmartWatts

4.3.3.1 Overview and Goals

The PowerAPI implementation *SmartWatts*[54] is a software-defined, self-calibrating power 'formula' designed for estimating power consumption of containers, processes, and VMs. It aims to address the shortcomings of static power models by using online model adaptation (sequential learning) and runtime performance counters. Unlike many academic models that require manual calibration or architecture-specific training, SmartWatts adapts automatically to the host system and workload.

4.3.3.2 Architecture and Metric Sources

SmartWatts is written in Python. Understanding the architecture of SmartWatts and its differences from other energy monitoring tools is crucial. Using HWPC, RAPL, and CPU process metrics, SmartWatts collects performance data. At runtime, it uses power models based on cgroups and perf events alone to estimate, for each resource $\text{res} \in \{\text{CPU}, \text{DRAM}\}$, the host power consumption $\hat{p}_{\text{res}}$ and the power consumption $\hat{p}_{\text{res}}(c)$ for all containers. SmartWatts uses $\hat{p}_{\text{res}}$ to continuously assess the accuracy of the managed power models $M_{\text{res},f}$ to ensure that estimated power consumption does not diverge from the RAPL baseline measurement $p_{\text{res}}^{\text{rapl}}$. When the estimation diverges beyond a configurable threshold ϵ_{res}, SmartWatts triggers a new online calibration process for the model. When the machine is at rest (e.g. after a reboot), this method is also used to isolate the static energy consumption. A simple architecture can be seen in Figure 4.3.

In practical terms, SmartWatts implements a server-side powermeter (referred to as *power meter*) that consumes input samples and produces power estimations accordingly. The power meter is responsible for power modelling, power estimation, and model calibration. In addition, a client-side sensor (referred to as *sensor*) is deployed as a lightweight daemon on all cluster nodes. The sensor is responsible for static power isolation, event selection, cgroups, and event monitoring. This separation allows for heterogeneous cluster nodes.

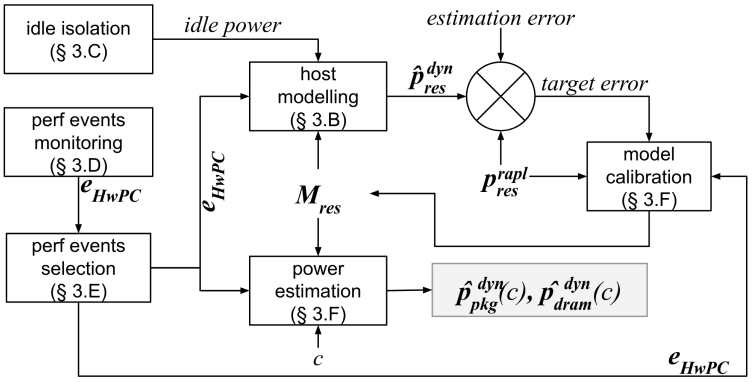


FIGURE 4.3: SmartWatts Architecture

4.3.3.3 Attribution Model

As discussed in the previous subsection, the SmartWatts attribution model does not use RAPL metrics, opting only for process metrics. SmartWatts separates host energy consumption into static and dynamic power consumption:

$$p_{res}^{rapl} = p_{res}^{static} + p_{res}^{dynamic} \tag{4.9}$$

Static power is estimated by periodically logging RAPL package and DRAM power consumption. The *median* value and the *interquartilerange* (IRQ) are gathered from the measurements to define the static host power consumption as

$$p_{res}^{static} := median_{res} - 1.5 \cdot IRQ_{res} \tag{4.10}$$

This approach is meant to filter out RAPL outliers.

Dynamic power is estimated by correlating the CPU frequency f and the raw metrics reported by HWPC:

$$\exists f \in F, \hat{p}_{res}^{dyn} = M_{res}^{f} \cdot E_{res}^{f} \tag{4.11}$$

where E_{res}^{f} denotes all *events*. The model M_{res}^{f} is built from *elastic net* regression applied on the last k samples. To ensure that all container power consumptions are linear with regards to global power consumption, positive inference coefficients are enforced, and the intercept (or *bias* term) is constrained within the range $[0, TDP]$.

HWPC metrics are dynamically chosen based on the list of available events exposed by the host's *Performance Monitoring Units* (PMU), essentially creating a custom model based on available metrics. Not all available metrics are used, and statistical analysis (Pearson coefficient) is applied to determine worthy candidates.

Container power consumption is estimated by applying the inferred power model M^f_{res} at the scale of the container's events $E^f_{res}(c)$, as seen in formula 4.12. In formula 4.13, the intercept i is distributed proportionally to the dynamic part of the consumption of c.

$$\exists f \in F, \forall c \in C, \hat{p}^{dyn}_{res}(c) = M^f_{res} \cdot E^f_{res}(c) \tag{4.12}$$

$$\forall c \in C, \tilde{p}^{dyn}_{res}(c) = \hat{p}^{dyn}_{res}(c) - i \cdot \left(1 - \frac{\hat{p}^{dyn}_{res}(c) - i}{\hat{p}^{dyn}_{res} - i}\right) \tag{4.13}$$

In theory, one can expect $\hat{p}^{dyn}_{res} \stackrel{!}{=} p^{dyn}_{res}$ if the model perfectly estimates the dynamic power consumption, but in practice, an error $\epsilon_{res} = \left| p^{dyn}_{res} - \hat{p}^{dyn}_{res} \right|$ occurs. Therefore, container power consumption is capped at:

$$\forall c \in C, \left\lceil \tilde{p}^{dyn}_{res}(c) \right\rceil = \frac{p^{dyn}_{res} \cdot \tilde{p}^{dyn}_{res}(c)}{\hat{p}^{dyn}_{res}} \tag{4.14}$$

This approach also allows calculating a confidence interval of the power consumption of containers by scaling down the observed global error:

$$\forall c \in C, \epsilon_{res}(c) = \frac{\tilde{p}^{dyn}_{res}(c)}{\hat{p}^{dyn}_{res}} \cdot \epsilon_{res} \tag{4.15}$$

In order to improve estimation accuracy, the following configurable parameters are used:

- CPU TDP in Watt (default: 125)
- CPU base clock in MHz (default: 100)
- CPU base frequency in MHz (default: 2100)
- CPU and DRAM error threshold in Watt (default: 2)
- Minimum number of samples required before attempting to learn a power model (default: 10)
- Size of the history window used to retain samples (default: 60)
- Measurement frequency in milliseconds (default: 1000)

4.3.3.4 Validation and Research Context

With RAPL being used as ground truth for dynamic power estimation model recalibration, it is important to note that the SmartWatts validation focuses on model accuracy when compared to RAPL values, rather than values obtained from an external source of power data. The SmartWatts validation evaluated the quality of power estimation under sequential and parallel workloads, the accuracy and stability of power models, and the overhead of the *sensor* component. Standard benchmarks like Stress-NG and NAS Parallel Benchmarks were used.

While no advanced statistical analysis was conducted, the validation shows that, for an error threshold of 5 Watts for CPU and 1 Watt for DRAM, power consumption can be reliably estimated with errors below 3 Watts and 0.5 Watts, respectively. The only case where the error exceeds the threshold is at the CPU idle frequency. Model stability is shown to improve significantly when lower recalibration frequencies are used. SmartWatts successfully reuses a given power model for up to 594 estimations, depending on frequency. Monitoring overhead is observed to be 0.333 Watts for the PKG domain and 0.030 Watts for the DRAM domain on average, at a measurement frequency of 2 Hz. The authors consider this overhead negligible.

4.3.3.5 Limitations and Open Issues

SmartWatts offers a compelling solution for dynamic, container-level power estimation through self-calibrating models based on performance counters. However, its applicability remains domain-specific. The central assumption is that RAPL, while accurate, is too coarse-grained for attributing power to individual containers or processes. This premise is debatable: RAPL offers low-overhead, high-frequency measurements and may be sufficient for many use cases, particularly in homogeneous or single-tenant systems. Whether SmartWatts' added complexity is justified depends on how fine-grained the attribution needs to be.

SmartWatts excels when more granular telemetry (e.g. perf events) is available and container-level attribution is critical. Yet its current implementation models only CPU and DRAM domains, limiting its ability to provide a comprehensive energy profile.

The design allows operators to supply hardware-specific values (e.g. CPU TDP), while falling back to sensible defaults. This improves usability without sacrificing model accuracy.

Finally, while SmartWatts' runtime calibration and dynamic event selection enhance adaptability, they introduce complexity. The event selection mechanism relies on statistical heuristics, which may not generalize well across systems. Moreover, under highly dynamic conditions, frequent recalibrations may affect stability.

In summary, SmartWatts is well-suited for environments requiring high-resolution attribution beyond RAPL's capabilities, but its scope, complexity, and assumptions warrant careful consideration depending on the target use case.

Chapter 5

Conclusion and Future Work

5.1 Summary of Findings

This thesis confirms that attributing energy consumption to containerized workloads remains an inherently complex and uncertain task. While existing tools can provide detailed measurements, none can achieve exact results: fundamentally, every method introduces estimation errors due to limitations in data sources and methodological assumptions. Measurement uncertainty is unavoidable and varies depending on the chosen metrics and approach.

The analyzed tools demonstrate significantly different design philosophies, largely reflecting their intended audiences and operational environments. For example, Kepler's earlier versions (v0.9.0) prioritized measurement granularity and resource-level accuracy, relying on eBPF instrumentation and hardware counters to achieve fine-grained attribution. In contrast, Kepler's latest version (v0.10.0) adopts a simplified, CPU-time-based attribution model that sacrifices accuracy in favor of deployability, security, and operational stability, suggesting a strategic shift.

The intended audience of a measurement tool is a critical factor influencing its architecture. Researchers and infrastructure engineers working on system optimization may demand maximal granularity and accuracy, accepting additional overhead and complexity. Developers, in contrast, often seek only high-level visibility into container-level energy consumption and may prefer black-box tools that abstract away implementation details. Cluster operators balance energy awareness with priorities such as security, availability, and maintainability; conditions that typically preclude privileged monitoring techniques like eBPF.

Across all tools, Intel's RAPL interface emerges as the most reliable and promising source of real-time energy telemetry for CPU and DRAM domains. Its millisecond-level update frequency theoretically enables highly granular measurements. However, most tools (including Kepler) fail to exploit this potential. Multi-second sampling intervals remain standard, reducing the fidelity of captured workload dynamics and limiting attribution accuracy, especially for heterogeneous workloads characterized by short-lived or bursty processes. Granular temporal analysis is thus identified as a key requirement for accurate energy attribution in complex environments. Fine-grained tracking better captures workload diversity and transient behavior but introduces computational overhead. Tools attempt to balance these competing concerns, but no clear consensus or universally optimal strategy emerges.

Another important trade-off lies between estimation and accuracy. While perfect energy attribution is unattainable, approximate models provide valuable insights even at lower precision. This thesis argues that estimation remains worthwhile, especially in multi-tenant and dynamic environments where detailed hardware telemetry is unavailable or incomplete.

Notably, all examined tools focus exclusively on CPU, RAM, and GPU energy consumption. No attempt is made to estimate power consumption of other system components such as storage devices or network interfaces. Extending measurement capabilities beyond the core compute elements presents a significant research opportunity. While such estimations are inherently less precise, even approximate visibility could enhance observability and inform optimization efforts.

In conclusion, energy attribution in containerized systems is a balancing act between detail and practicality. Existing tools demonstrate that both high-accuracy and operational simplicity are valid design goals, serving different user groups and system environments. RAPL remains central to accurate measurement, but practical deployment constraints often limit its effective use. Achieving reliable and actionable power monitoring requires careful architectural decisions, guided by the specific needs of the intended audience and the realities of the deployment environment.

5.2 Critical Reflection

5.2.1 Methodological Reflection

This thesis adopts a purely analytical approach, centered on literature review, code analysis, and architectural evaluation of existing container-level energy attribution tools, as well as potential sources of information. This methodological choice aligns with the intended scope of the project: as a VT2-level research work, the thesis was designed to provide a theoretical foundation for subsequent practical development.

While the thesis proposes concrete recommendations for future tool design (see § 5.3), it deliberately refrains from developing a full tool architecture or implementation. Instead, the findings are intended to inform such efforts in future research, particularly within the scope of the author's upcoming master's thesis.

While empirical validation through experimental benchmarking was deliberately omitted to prioritize tool coverage and architectural analysis, this remains a methodological limitation. Validation insights from existing literature were integrated where available.

The focus on Linux and Kubernetes environments further narrows the applicability of the findings. Alternative infrastructures, proprietary tools or telemetry sources (e.g. OEM-specific BMC energy reporting interfaces) were not explored in depth. This reflects both the author's area of expertise and the practical relevance of Kubernetes in modern cloud environments but limits generalizability.

Despite these limitations, the thesis's core strengths lie in its detailed inspection of source metics, tool architectures and source code, combined with a practical understanding of Kubernetes-based deployments. This approach allowed the identification of undocumented behaviors, implementation inconsistencies, and unaddressed design trade-offs in tools such as Kepler, Scaphandre, and SmartWatts.

In summary, while empirical validation and cross-environment generalization are lacking, the thesis successfully fulfills its role as a theoretical exploration of container energy attribution, providing a solid analytical foundation for future tool development and evaluation.

5.2.2 Tool Adoption in Real-World Systems

A critical reflection of container-level energy attribution tools reveals significant barriers to their adoption in production environments. Chief among these are security concerns and the need for privileged access. Tools that rely on kernel-level instrumentation, such as eBPF or `perf_event_open`, often require elevated permissions, introducing potential security risks. For most cluster operators, energy monitoring remains a secondary concern compared to reliability, availability, and security. This limits the practical deployment of high-precision tools in real-world systems.

Operational simplicity frequently outweighs measurement accuracy. Tools with complex configurations, hardware-specific dependencies, or non-standard export interfaces are typically avoided, even if they promise higher measurement fidelity. In practice, both industry users and academic researchers often treat energy monitoring tools as black boxes. This is evident from the widespread use of tools like Kepler and Scaphandre without detailed validation or configuration tuning. While these tools offer configurability, leveraging it requires substantial technical understanding, which many users lack or are unwilling to invest.

As a result, most energy monitoring tools prioritize usability over accuracy. Prometheus integration, simple deployment (e.g. via Helm charts), and minimal security concerns are often deemed more important than methodological rigor. No tool analyzed in this thesis explicitly targets researchers seeking maximal measurement accuracy; instead, tools implicitly address operators who require straightforward observability solutions.

Finally, tool design often fails to recognize the fundamentally different needs of developers, operators, and researchers. Currently, no single tool effectively caters to all these audiences. This segmentation of user requirements, combined with practical deployment constraints, helps explain why most existing tools settle for operational simplicity at the expense of measurement accuracy.

5.2.3 Transparency, Trust, and Black-Box Measurement

A fundamental challenge in container-level energy attribution is the reliance on inherently opaque measurement interfaces. Critical telemetry sources such as Intel RAPL, NVIDIA NVML, and platform-level BMC sensors provide essential power and energy data, yet their internal operation and measurement scopes remain poorly documented. For instance, both Scaphandre and Kepler developers acknowledge uncertainty regarding the exact coverage of RAPL domains, most notably `PKG` and `PSys`. This lack of clarity complicates both tool implementation and the interpretation of reported energy metrics.

Black-box measurement interfaces hinder the development of accurate and explainable energy monitoring tools. When the underlying telemetry mechanisms are closed, tool developers are forced to make assumptions which propagate into the attribution models and directly impact reported results. Without visibility into how energy

counters are computed or which hardware components are included, users cannot fully trust the reported energy consumption data, nor can they debug unexpected results.

To address this, future energy monitoring frameworks should:

- Clearly document all assumptions related to telemetry sources and attribution models.
- Provide visibility into the source and scope of every reported metric.
- Encourage open standards for energy telemetry, advocating for greater transparency in RAPL, NVML, and similar interfaces.

5.2.4 Energy Attribution Philosophies

The analysis confirms that energy attribution models (container-centric, shared-cost, and residual modeling) reflect fundamentally different perspectives. What is considered a "fair" distribution depends on the user's priorities: developers, operators, or researchers will each favor different approaches.

Current tools often make implicit attribution decisions, especially regarding idle power and system processes, without clear documentation. This obscures how reported metrics should be interpreted.

To improve transparency and usability, future tools should:

- Clearly document their attribution model.
- Where feasible, allow users to choose between attribution strategies.
- Make residual power explicit rather than hidden in shared costs.

5.3 Recommendations for Future Tool Development

5.3.1 Towards Maximum-Accuracy Measurement Tools

Future container-level monitoring tools should prioritize temporal resolution, metric flexibility, and modular attribution models to maximize measurement accuracy. Based on the findings of this thesis, the following design principles are recommended:

High-Resolution Hardware Metrics RAPL-based measurements should support sub-second sampling intervals configurable by the user. Intervals as low as 50 milliseconds (close to RAPL's practical resolution limit) would enable significantly finer-grained power attribution, especially in heterogeneous or bursty workloads. Critically, power readings should be attributed directly upon collection, avoiding fixed aggregation cycles that dilute temporal precision and introduce attribution inaccuracies.

Decoupled Metric Handling Metric collection loops should differentiate between high-frequency (e.g. RAPL, eBPF) and low-frequency (e.g. Redfish, IPMI) sources.

Separating high-priority, high-frequency metrics from slower telemetry sources minimizes performance overhead and maximizes the utility of each metric type.

Multi-Metric Integration Tools should support combining diverse telemetry sources, such as RAPL, Redfish, ACPI, and BMC, in a coherent manner. Coarse-grained metrics (e.g. Redfish node-level power) can be fused with fine-grained metrics (e.g. RAPL domain-level power) to interpolate or validate measurements. However, care must be taken to preserve the distinction between direct measurements and model-based estimations when combining such sources.

User-Configurable Estimation Modules Modular estimation frameworks should be employed for subsystems lacking direct telemetry, such as storage devices or network interfaces. Default models can provide reasonable estimates based on automatic device detection (e.g. storage device type, or link speed for network interfaces). However, advanced users should be able to override idle, maximum, and typical power values to refine model accuracy.

Selectable or configurable Attribution Models Energy attribution should support multiple modeling approaches, ideally selectable at runtime. Examples include container-centric models, shared-cost models, or hybrid methods. Importantly, idle and system-level energy consumption should be accounted for explicitly, not implicitly merged into container totals, improving transparency and accuracy.

Self-Calibration Support Tools should offer automated or semi-automated calibration methods, such as idle power calibration or workload-based calibration inspired by Kavanagh et al. [12]. Where possible, standardized interfaces for integrating external measurement devices should be considered, enabling users to validate or refine energy models via external power meters.

Standards-Based Implementation Wherever feasible, tools should adhere to standardized system interfaces such as the Linux `powercap` framework for RAPL access, avoiding proprietary solutions. This facilitates long-term maintainability and eases deployment across heterogeneous environments.

In summary, a high-accuracy monitoring tool must prioritize both technical rigor and architectural flexibility. Drawing from design strengths observed in Kepler, KubeWatt, Scaphandre, and SmartWatts, future tools should offer high-resolution measurements, modular estimation, and transparent energy attribution as core design objectives.

5.3.2 Addressing Missing Domains: Disk, Network, and Others

No current container-level energy monitoring tool provides direct measurements or estimations for storage devices or network interfaces. Nevertheless, for a comprehensive understanding of node-level energy consumption, these components should not be neglected.

Modular Estimation Frameworks Future tools should include optional, modular estimation models for disks and network interface controllers (NICs). Such models could leverage existing system metrics such as I/O request counts, throughput

rates, or link speeds as input signals. For example, storage energy estimation could differentiate between SSDs and HDDs based on device identification, using I/O operations as a proxy for activity levels. Similarly, NIC models could base estimations on transmitted and received data volumes or link activity states.

Inherent Accuracy Limitations These estimations will inevitably remain less accurate than direct telemetry from hardware sensors. However, including such models can enhance the completeness of node-level energy consumption analysis, particularly in environments where disks and NICs constitute non-trivial portions of total power draw.

Residual Energy Utilization In cases where total node power is known (e.g. via Redfish or BMC sensors) and major contributors like CPU and memory are directly measured, residual energy (the unaccounted portion) could be partially attributed to storage and networking components. However, reliance on residual energy must be approached cautiously, as it risks compounding measurement and attribution errors.

Configurability Estimation models should remain user-configurable. While default values enable ease of use for casual users, advanced users should be able to fine-tune idle power, maximum power, and activity-to-power correlation parameters to improve estimation accuracy.

In summary, although disk and network power estimations are inherently imprecise, including them in a modular and configurable manner would significantly enhance the practical value of container-level energy monitoring tools.

5.3.3 Balancing Accuracy and Overhead

The pursuit of maximum measurement accuracy inevitably increases monitoring overhead. Future tools should address this trade-off by offering distinct operational modes, allowing users to select between accuracy and resource efficiency based on their specific needs.

'Precision' Mode In this mode, all available telemetry sources and fine-grained attribution models should be enabled. High-frequency sampling intervals, detailed container-level breakdowns, and optional estimations for secondary components (e.g. disks, NICs) provide maximal measurement detail. This mode is intended for research, validation, or auditing scenarios where energy transparency is prioritized over runtime performance.

'Lightweight' Mode Conversely, a lightweight mode should disable high-frequency probes, omit low-relevance subsystems, and focus on core power consumers such as CPU and memory. Sampling intervals can be relaxed, and coarse-grained metrics prioritized. This configuration is suitable for production environments where minimizing monitoring overhead is critical.

Mode Selection Not all environments or users require maximum accuracy. By providing predefined operational modes, tools can adapt to a wide range of use cases

without forcing users to manually configure every parameter. However, manual overrides should remain possible for expert users seeking fine control.

In summary, supporting both 'precision' and 'lightweight' modes allows monitoring tools to serve diverse operational contexts without compromising on flexibility or usability.

5.3.4 Supporting Multiple User Roles and Needs

Container-level energy monitoring tools must accommodate a diverse range of users, each with distinct goals and expectations. A future-proof tool should address these needs through architectural modularity, clear defaults, and extensive documentation.

Developers Developers typically seek simple, per-container energy consumption totals to guide software optimization. Their focus is on understanding the energy impact of specific applications or containers, without concern for the idle power waste or baseline energy consumption of the broader system. Minimal setup complexity and straightforward metric outputs are priorities for this user group.

Operators Infrastructure operators require system-wide energy observability, not limited to individual containers. Their goals include identifying idle energy waste, optimizing resource utilization, and avoiding performance degradation due to throttling or resource contention. Operators value transparency, stability, and actionable insights across the entire infrastructure stack.

Researchers Researchers demand the highest levels of accuracy, configurability, and architectural transparency. They require detailed documentation of attribution models, known limitations, and telemetry sources, alongside access to raw metrics and calibration options. Flexibility and reproducibility are critical requirements for this audience.

Serving All Audiences To accommodate these varied needs, monitoring tools should adopt a modular architecture with:

- Sensible, production-ready defaults for black-box usability.
- Optional advanced configuration layers for expert users.
- Clear and comprehensive documentation describing methods, assumptions, and limitations.

Recognizing that many users will treat the tool as a black box, default configurations must produce reasonable, usable results without requiring manual tuning. However, advanced users should retain the ability to inspect, customize, and extend the tool's behavior.

In summary, serving developers, operators, and researchers simultaneously requires balancing simplicity, flexibility, and transparency within the tool's design.

5.3.5 Energy Metrics for Virtualized Environments

Energy attribution within virtualized environments, particularly for Kubernetes clusters running inside virtual machines (VMs), remains a challenge. Existing monitoring tools primarily target bare-metal deployments, leaving a significant gap in energy observability for cloud-based and virtualized infrastructures.

Existing Approaches Two conceptual approaches have been explored:

- **Scaphandre's QEMU Passthrough:** Implements a basic export mechanism by writing host-side energy metrics to a virtual file system accessible by guest VMs running an identical instance of Scaphandre. This approach is functional but but limited, since input and output metrics must correlate to serve identical instances.
- **Kepler's Hypercall Concept:** Proposes using hypercalls as a mechanism for host-to-guest metric transfer. However, this concept remains unimplemented.

Future Directions Future monitoring tools should prioritize the development of standardized telemetry export mechanisms to enable accurate energy monitoring within VMs. Potential approaches include:

- Hypervisor-supported hypercalls specifically designed for energy metrics.
- Virtio-based APIs to expose host-side telemetry directly to guest VMs.
- Enhanced QEMU or container runtime interfaces capable of exporting power data.

Relevance Given the prevalence of virtualized infrastructure in modern cloud environments, particularly for managed Kubernetes platforms, solving this problem would significantly broaden the applicability and adoption of energy observability solutions.

In summary, enabling reliable host-to-VM energy metrics passthrough represents a critical development priority for future container-level energy monitoring tools.

5.3.6 Standardization and Hardware Vendor Transparency

Measurement accuracy in container-level energy monitoring is fundamentally limited by the availability and transparency of hardware telemetry. Addressing these constraints requires both industry-wide standardization efforts and increased openness from hardware vendors.

Hardware Vendor Responsibility Vendors should provide native power telemetry for additional system components, such as network interface cards (NICs), storage devices, and peripheral subsystems. These metrics should be accessible via standardized, open interfaces to facilitate direct measurement and reduce reliance on model-based estimations.

Expanding Telemetry Standards Existing interfaces like Redfish, which currently expose node-level power data, could be extended to include per-component power

reporting. Similarly, the adoption of open, vendor-neutral standards for exposing telemetry at the subsystem level would significantly enhance energy observability.

RAPL Transparency The lack of public documentation regarding Intel's Running Average Power Limit (RAPL) interface remains a barrier to fully understanding and validating the reported power domains. Vendors should disclose the internal structure and calculation methods of such telemetry systems to resolve current 'black-box' concerns identified by tool developers, including Kepler and Scaphandre contributors.

In summary, improving measurement precision at the workload level depends not only on software design but also on cooperation from hardware manufacturers and industry standards bodies. Transparent and standardized telemetry interfaces represent a critical enabler for future progress in energy observability.

5.4 Broader Research and Industry Opportunities

The advancement of energy observability in containerized environments extends beyond tool development. Broader collaboration across industry stakeholders and research communities is essential to drive progress.

Role of Standards Organizations Organizations such as the Cloud Native Computing Foundation (CNCF) and Kubernetes Special Interest Groups (SIGs) could play a central role in promoting standard APIs for energy metric collection and dissemination within cloud-native environments. Establishing best practices and reference implementations would encourage adoption and consistency across tools.

Hardware Manufacturer Involvement Hardware vendors are positioned to directly influence the quality and availability of energy telemetry data. By exposing accurate and accessible power metrics across all major system components, manufacturers can enable precise energy measurement without reliance on coarse or estimated models. Collaboration with open-source communities could further support development of vendor-agnostic solutions.

Sustainable Cloud Computing Energy observability should be recognized as a foundational component of sustainable cloud computing. Accurate energy measurement at the workload level enables informed optimization decisions, supports regulatory compliance, and advances corporate sustainability goals. As such, energy transparency should be integrated into both research agendas and industry roadmaps for cloud infrastructure development.

In summary, advancing energy observability requires coordinated efforts spanning tool developers, standards bodies, hardware vendors, and sustainability-focused initiatives.

5.5 Closing Remarks

Energy consumption measurement at the container or workload level remains a complex and evolving challenge. This thesis has identified key architectural and

methodological features that future monitoring tools should incorporate to enhance measurement accuracy, architectural flexibility, and practical usability.

However, progress in this field is constrained not only by technical trade-offs (such as balancing accuracy against monitoring overhead) but also by external limitations. Chief among these are the lack of transparent hardware telemetry, incomplete standardization of power reporting interfaces, and the absence of established methodologies for energy attribution in virtualized environments.

The recommendations presented in this chapter are intended to guide both tool developers and researchers. By integrating high-resolution hardware metrics, modular estimation frameworks, user-configurable attribution models, and standardized interfaces, future tools can advance the state of energy observability in containerized infrastructures.

Ultimately, energy-aware computing practices will become increasingly relevant as the industry shifts towards sustainable cloud operations. Improved energy transparency at workload granularity represents a critical foundation for enabling these efforts.

Appendix B

Powerstack

Implementation of an energy monitoring environment in Kubernetes

Zurich University of Applied Sciences

Department School of Engineering

Institute of Computer Science

SPECIALIZATION PROJECT 1

Powerstack: Implementation of an energy monitoring environment in Kubernetes

---

Author:
Caspar Wackerle

Supervisors:
Prof. Dr. Thomas Bohnert
Christof Marti

Submitted on
JANUARY 31, 2025

Re-edited on
January 31, 2026

Study program:
Computer Science, M.Sc.

Abstract

Energy efficiency in cloud computing has become a critical concern as data centers consume an increasing share of global electricity. This thesis investigates energy consumption at the container and node level in Kubernetes-based infrastructures, using KEPLER (Kubernetes-based Efficient Power Level Exporter) to monitor and analyze power consumption in a controlled test environment.

A bare-metal Kubernetes cluster was deployed on three identical servers, configured using K3s for lightweight orchestration and managed through Ansible for full automation. The entire system was designed to be fast to deploy, highly reproducible, and adaptable to different hardware environments. Configurations were centralized for easy reusability in future projects, ensuring that modifications could be made with minimal effort. Prometheus and Grafana were integrated to collect and visualize KEPLER's real-time energy consumption metrics. A series of controlled benchmarking experiments were conducted to stress CPU, memory, disk I/O, and network I/O, assessing KEPLER's accuracy in reporting power usage under varying workloads.

The results indicate that KEPLER credibly tracks workload-induced power variations at the CPU package level, though inconsistencies arise in non-CPU power domains. High idle power consumption was observed at the node level, suggesting that infrastructure energy efficiency must account for static consumption beyond dynamic workloads.

This thesis provides a foundation for further research into energy-efficient Kubernetes environments, including improving KEPLER's accuracy, extending workload profiling, and exploring automation-driven energy optimization strategies. The modular and automated deployment architecture ensures that the findings and methodologies can be readily adapted for use in other energy-related cloud research projects.

The accompanying source code for this thesis, including all deployment and automation scripts, is available in the **PowerStack**[1] repository on GitHub.

Chapter 1

Introduction and Context

1.1 Significance of Energy Efficiency in Cloud Computing

Cloud computing has reshaped how computing resources are provisioned and consumed, offering efficiency gains through large-scale resource sharing. While economic benefits (such as reduced operational costs and improved scalability) are widely recognized, energy efficiency and environmental impact are equally important considerations. By enabling higher utilization of shared infrastructure, cloud computing can reduce energy waste and support global sustainability objectives.

The rapid growth of cloud adoption has transformed it into a central component of global IT infrastructure. Hyperscalers such as Amazon Web Services, Google Cloud, and Microsoft Azure are now major consumers of electrical power, drawing increasing attention from policymakers and environmental organizations. Although cloud providers have made significant investments in renewable energy procurement, the use of green power alone does not guarantee efficient energy utilization at workload level.

Technological advancements have continuously improved the energy efficiency of data centers, with modern facilities achieving Power Usage Effectiveness (PUE) values close to 1. However, PUE reflects facility-level efficiency and does not capture how effectively energy is used by individual workloads. Even at a theoretical PUE of 1, substantial waste may occur if computational resources are underutilized. This underscores the need to examine workload-level energy efficiency.

Containers, as a lightweight virtualization technology, improve resource density and often deliver better energy efficiency than traditional virtual machines (VMs). However, they also introduce new challenges for accurate energy measurement. Granular monitoring becomes more complex in containerized environments (especially in Kubernetes) due to dynamic resource allocation, scheduling, and autoscaling mechanisms.

Despite the importance of this topic, research on energy efficiency in cloud-native systems remains limited. While data center operations have been extensively optimized and green-coding practices are increasingly emphasized, the energy efficiency of Kubernetes clusters is comparatively underexplored. Addressing this gap is essential for aligning economic and environmental goals.

1.2 The Need for Energy-Efficient Kubernetes Clusters

Kubernetes has become the de facto standard for container orchestration, managing large-scale, distributed workloads. Its sophisticated features (dynamic scheduling, autoscaling, and distributed resource management) enable high performance and operational flexibility. However, these same features complicate energy measurement and optimization.

In many deployments, Kubernetes clusters run on virtual machines to simplify infrastructure management, adding an additional abstraction layer and making energy attribution more difficult. Achieving energy-efficient Kubernetes clusters therefore requires robust measurement techniques that can translate raw energy data into actionable insights. Such measurements form the basis for systematic optimization efforts.

Given the growing energy footprint of cloud infrastructures and the limited research on energy efficiency in Kubernetes, this remains a pressing and timely research area. By addressing this gap, the present work contributes to the broader objective of sustainable cloud computing.

1.3 Objectives and Scope of this Thesis

1.3.1 Context

This thesis is part of the Master's program in Computer Science at the Zurich University of Applied Sciences (ZHAW) and represents the first of two specialization projects. The current project (VT1) focuses on the practical implementation of a test environment for energy-efficiency research in Kubernetes clusters. The subsequent project (VT2) will examine theoretical foundations and methodological approaches to measuring and improving energy efficiency in such environments.

The work builds upon previous projects on performance optimization and energy measurement. EVA1 addressed topics such as operating-system tooling, statistical foundations, and eBPF, while EVA2 explored energy measurement across hardware, firmware, and software layers. These foundations inform the present thesis but will not be revisited in detail.

1.3.2 Scope

This thesis focuses on the practical implementation of a test environment, excluding detailed theoretical analysis and extensive literature review. The primary goal is to document the creation of a reliable and reproducible environment that supports future research on energy efficiency in Kubernetes clusters.

The work builds upon prior activities carried out in EVA1 and EVA2, leveraging existing knowledge while extending the research focus. Fundamental concepts from these earlier projects are assumed as background knowledge and are therefore not revisited in detail.

The EVA1 presentations introduced essential principles related to Linux performance monitoring, system optimization, and a conceptual understanding of eBPF. While these concepts play an important role in this research, they are not re-explained here,

as the emphasis of this thesis lies on their practical application within an energy-efficient Kubernetes cluster.

Similarly, the EVA2 presentations established foundational knowledge for understanding energy-consumption measurement across hardware, firmware, and software layers. Topics such as CPU power states (ACPI, C-states, P-states), Intel CPU performance-scaling drivers, and Intel RAPL for power monitoring and control are considered essential prerequisites but are not explicitly covered again in this thesis.

By building on these existing foundations, this thesis narrows its scope to the investigation of energy efficiency at the Kubernetes cluster level, incorporating relevant techniques from prior research where appropriate.

1.3.3 Objectives

The main objective is to design and implement a test environment that enables:

- Analysis of key parameters affecting energy efficiency in Kubernetes clusters.
- Reliable and consistent experimentation.
- Reproducibility and automation in deployment and configuration.

The resulting environment will form the basis for subsequent research projects.

1.3.3.1 Parameters for Analysis

The project aims to reuse established tools and components where feasible. The parameters to be analyzed include:

- CPU utilization and energy consumption.
- Memory usage and its impact on power draw.
- Disk I/O and storage-related power consumption.

Additional parameters may be incorporated following further evaluation.

1.3.3.2 Data Integrity and Persistence

Ensuring data integrity and persistence is critical for reliable analysis. Key requirements include:

- Persistent storage that survives system shutdown.
- A unified data store accessible by all nodes.
- Data retention across Kubernetes cluster reinstallations.
- The ability to power down unused worker nodes without data loss.

1.3.3.3 Reproducibility and Automation

Reproducibility and automation are additional goals aimed at improving research efficiency. Benefits include:

- Simplified recovery from misconfiguration through rapid redeployment.
- Reduced troubleshooting time.
- Improved stack cleanliness by eliminating residual configurations.

1.3.3.4 Security

Security, while not a primary focus, will be addressed by implementing basic best practices. Key measures include:

- Use of encrypted passwords.
- Adherence to standard Kubernetes security best practices.
- Minimization of potential vulnerabilities through careful configuration.

1.3.4 Use of AI Tools

During the writing of this thesis, *ChatGPT*[63] (Version 4, OpenAI, 2024) was used as an auxiliary tool to improve efficiency in documentation and technical writing. Specifically, it assisted in:

- Structuring and improving documentation clarity.
- Refining descriptions of code and technical implementations.
- Beautifying and formatting smaller code snippets.
- Assisting in LaTeX syntax corrections and debugging.

All AI-generated text was critically reviewed, edited, and adapted to the specific context of this thesis. **ChatGPT was not used for literature research, conceptual development, methodology design, or analytical reasoning.** The core ideas, analysis, and implementation details were developed independently.

1.3.5 Project Repository

All code, configurations, and automation scripts developed for this thesis are publicly available in the PowerStack[1] repository on GitHub. The repository includes Ansible playbooks for automated deployment, Kubernetes configurations, monitoring-stack setups, and benchmarking scripts. This ensures full reproducibility of the test environment and facilitates further research or adaptation for similar projects.

Chapter 2

Architecture and Design

2.1 Overview of the Test Environment

The test environment consists of a Kubernetes cluster deployed on three bare-metal servers located in a university datacenter. The servers are identical in their hardware specifications and are connected through both a private network and the university network. This setup enables complete remote management and ensures direct communication between the servers for Kubernetes workloads. A detailed description of the hardware and network topology is provided below. Figure 2.1 illustrates the overall architecture and network configuration.

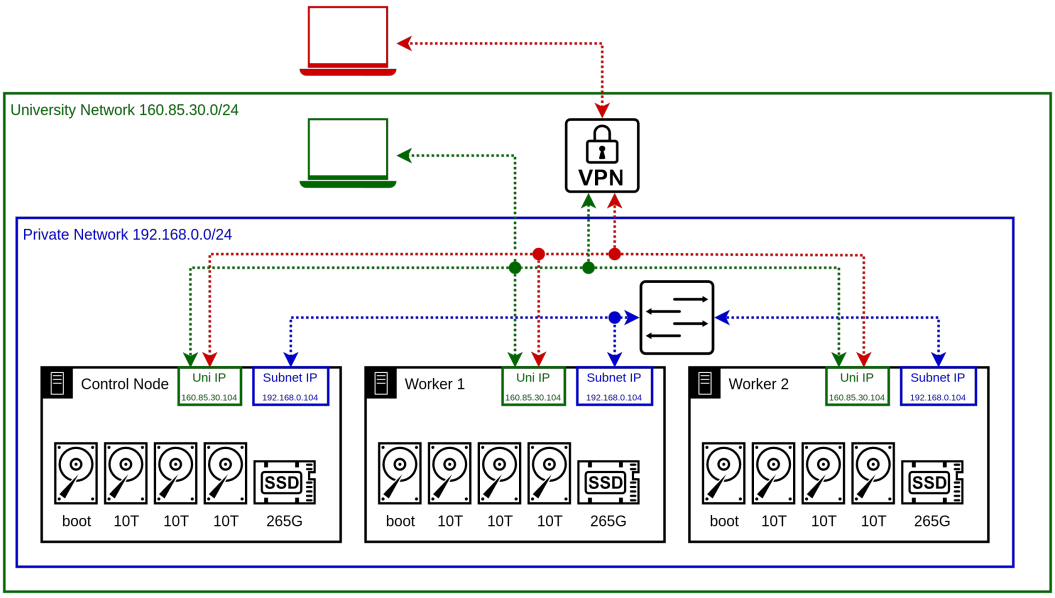


FIGURE 2.1: Physical Infrastructure Diagram

2.1.1 Hardware and Network

2.1.1.1 Bare-Metal Servers

The cluster is built using three identical Lenovo ThinkSystem SR530 servers, each equipped with the following hardware:

- CPU: 1 × Intel(R) Xeon(R) Bronze 3104 @ 1.70 GHz, 6 cores.

- Memory: 4 × 16 GB DDR4 DIMMs (total 64 GB RAM).
- Storage:
  - 2 × 32 GB M.2 SATA SSD (operating-system boot drive).
  - 1 × 240 GB 6 Gbps SATA 2.5″ SSD (persistent storage).
  - 3 × 10 TB 7200 RPM 12 Gbps SAS 3.5″ HDD (bulk storage).
- Power: Dual redundant power supplies.
- Cooling: 4 of 6 possible fans installed.
- Firmware:
  - BMC Version: 8.88 (Build ID: CDI3A4A)
  - UEFI Version: 3.42 (Build ID: TEE180J)
  - LXPM Version: 2.08 (Build ID: PDL142H)

Each server includes a Lenovo XClarity Controller (BMC) for remote management. Access via the BMC IP provides out-of-band management and monitoring capabilities.

2.1.1.2 Network Topology

The servers are connected using two distinct networks:

- **Private Network:** Each server has a private IP address (192.168.0.104–192.168.0.106), enabling direct, high-speed communication between nodes. This reduces load on the university network and improves performance for intra-cluster Kubernetes workloads.
- **University Network:** Public-facing IP addresses (160.85.30.104–160.85.30.106) allow access from within the university network, with external access available via VPN.

Note: Detailed switch and gateway configurations are maintained by the university IT department and are beyond the scope of this document.

2.2 Key Technologies

2.2.1 Ubuntu

Ubuntu was selected as the operating system primarily due to the author's familiarity with it. Additionally, it was already installed on the servers when they were received, which reduced setup time and complexity. While other Linux distributions are optimized specifically for Kubernetes, using a well-known distribution ensured a smoother initial configuration process.

2.2.2 Bare-Metal K3s

Deploying Kubernetes directly on bare-metal servers (without using a hypervisor or virtual machines) was a fundamental design decision to ensure direct access to hardware-level data. This enables Kubernetes components and monitoring tools to interact with the underlying hardware more effectively, an essential requirement for accurate energy-consumption measurements.

K3s was selected for several reasons:

- It is lightweight and therefore suitable for lower-powered servers while potentially reducing overall energy consumption.
- Despite its minimal footprint, it remains fully compatible with upstream Kubernetes, allowing standard resources and configurations to be used without modification.
- K3s includes optimizations for ARM-based systems, making it a convenient choice for homelab environments and flexible future deployments.
- The author had prior experience with K3s and Rancher, enabling a faster and more reliable setup.

2.2.3 Ansible, Helm, `kubectl`

For automation and deployment, Ansible and Helm were chosen. Helm and `kubectl` are standard tools for managing Kubernetes applications and resources, offering broad community support and extensive documentation.

Ansible was selected for its flexibility and its simplicity when managing server configurations across multiple nodes. Its agentless approach (requiring only SSH and Python on the target machines) makes it particularly suited for managing bare-metal servers.

2.2.4 Kube-Prometheus Stack

The Kube-Prometheus stack was chosen because it is the de facto standard for monitoring in Kubernetes environments. The project is mature, feature-rich, and integrates seamlessly with Kubernetes components. Installation and configuration via Helm are straightforward, and the wide availability of community resources simplifies troubleshooting.

2.2.4.1 Prometheus

Prometheus was selected as the primary monitoring tool due to its strong integration with Kubernetes. While it provides extensive capabilities, it also introduces overhead and is not well suited for sub-second or low-second intervals, as typical scrape intervals are longer. For use cases focused on container orchestration, where lifetimes tend to be longer, this limitation is acceptable.

2.2.4.2 Grafana

Grafana was included for its ability to provide intuitive, customizable visualizations of metrics collected by Prometheus. It enables efficient interpretation of complex data through dashboards and visual representations, making it a valuable component of the monitoring stack.

2.2.4.3 AlertManager

AlertManager is bundled with the Kube-Prometheus stack and is used to route and manage alerts generated by Prometheus. Although AlertManager was not utilized in this project, its presence is beneficial for potential future extensions, particularly in production-like scenarios involving alerting and incident response.

2.2.5 Kepler

2.2.5.1 Purpose of Kepler

KEPLER[49], the *Kubernetes-based Efficient Power Level Exporter*, is a project focused on measuring energy consumption in Kubernetes environments. It provides detailed power-consumption metrics at the process, container, and pod levels, addressing an increasingly important need for energy-efficient cloud computing.

With cloud providers and enterprises facing growing pressure to improve energy efficiency, Kepler offers a practical solution. By enabling detailed, near-real-time measurement of power usage, it bridges the gap between high-level infrastructure metrics and workload-specific energy data. This ability to attribute energy consumption to individual components makes Kepler a valuable tool for advancing energy-efficient Kubernetes clusters.

2.2.5.2 Limitations of Kepler

Despite its potential, Kepler exhibits several limitations in the context of this project:

- **Active development:** Kepler is still under active development, meaning that features and APIs may change frequently. Documentation is limited, and community support for troubleshooting is still evolving.
- **Complexity:** Kepler is a large system with a complex architecture. Adapting it beyond basic configuration requires a deep understanding of its internal structure, making custom changes or enhancements challenging without substantial expertise.

Although Kepler is not without drawbacks, it remains the most promising available solution for measuring energy consumption in Kubernetes environments. Consequently, this thesis places significant emphasis on evaluating Kepler's capabilities and identifying potential areas for improvement.

2.3 Architecture and Design

2.3.1 Kubernetes Cluster Design

The Kubernetes cluster is deployed on three bare-metal servers running Ubuntu. One server is designated as the control-plane node, while the remaining two serve as worker nodes. For simplicity and in line with the project scope, no high-availability (HA) configuration is used. The servers communicate via their internal IP addresses, ensuring direct node-to-node communication without traversing external networks.

All Kubernetes control-plane components such as the API server, controller manager, and scheduler run exclusively on the control-plane node, while workloads are distributed across all nodes. Figure 2.1 provides an overview of the system architecture, including major components and data flow.

2.3.2 Persistent Storage

Persistent storage is provided using the spare SSD installed in the control-plane server. A partition on the SSD is created, formatted with the BTRFS filesystem, and mounted as the primary storage location. The control-plane node hosts an NFS server, and the worker nodes mount the corresponding NFS share to access the shared storage.

This centralized approach was chosen because the control-plane node is guaranteed to remain powered on throughout all experiments, making distributed storage solutions such as CEPH unnecessary for this project.

Within the NFS share, separate directories are allocated for Prometheus and Grafana data. Persistent volumes (PVs) and persistent volume claims (PVCs) are created for each service. The size of the PVs is configurable at installation time, providing flexibility for future storage requirements.

2.3.3 Monitoring Architecture

The monitoring stack is deployed using the `kube-prometheus-stack` Helm chart. This stack bundles Prometheus, Grafana, and AlertManager, providing a complete monitoring and visualization solution for Kubernetes environments. Prometheus scrapes metrics from Kepler and key Kubernetes endpoints (such as the kubelet API) at configurable intervals. Grafana connects to Prometheus to visualize these metrics through customizable dashboards.

2.3.4 Metrics Collection and Storage

Kepler generates energy-related metrics by collecting data from several sources:

- **Hardware-level metrics:** Using `eBPF` and kernel tracepoints to gather low-level data such as CPU cycles and cache misses.
- **Power-related metrics:** Obtained through RAPL (Running Average Power Limit) and IPMI (Intelligent Platform Management Interface) to measure CPU and platform-level energy consumption.

- **Container-level metrics:** Retrieved via the Kubernetes kubelet API, which exposes cgroup resource usage for running containers and pods.

Kepler aggregates these data sources, computes power-consumption metrics, and exposes them in a Prometheus-compatible format. Prometheus scrapes the metrics and stores them as time-series data on persistent storage, enabling detailed analysis of resource-usage patterns over time.

Chapter 2.4 provides a brief overview of the Kepler architecture, with a focus on metrics collection and generation. Figure 2.2 illustrates the information flow across the monitoring stack.

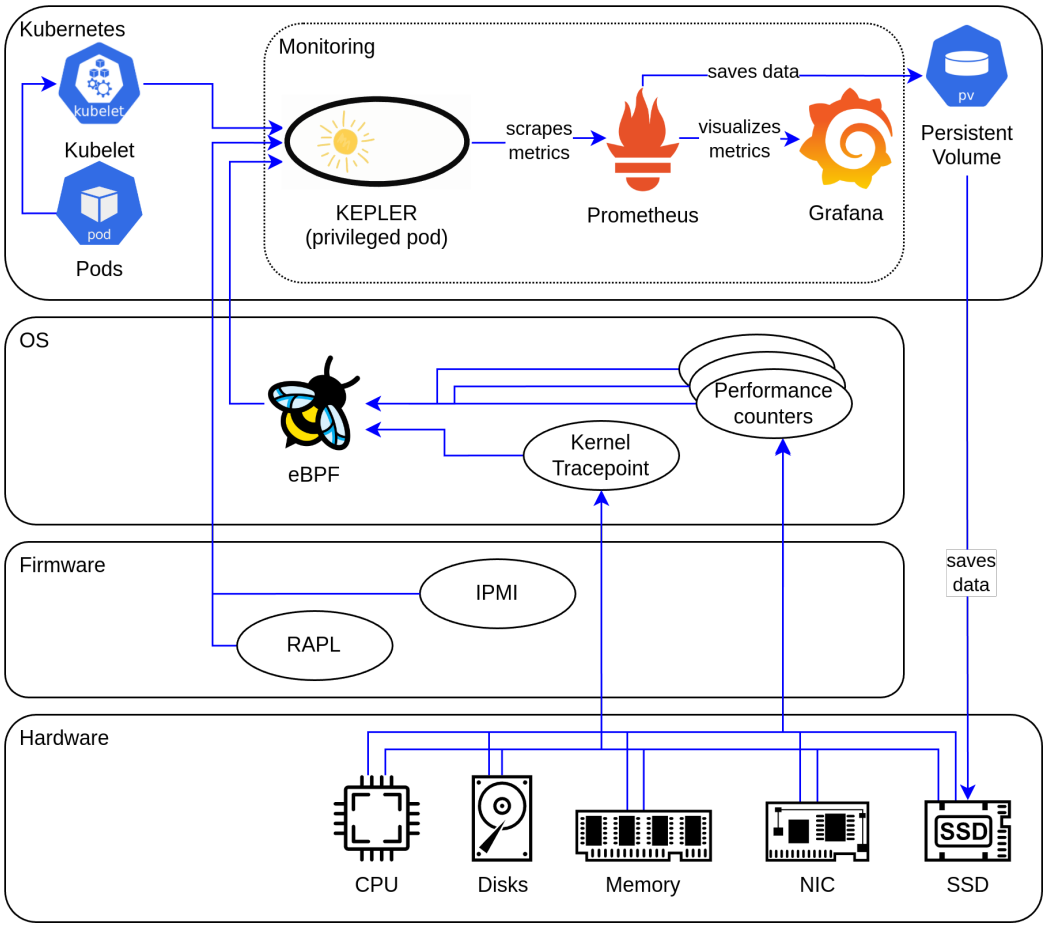


FIGURE 2.2: Monitoring data flow diagram of the entire stack

Given the substantial research and practical implementation already accomplished by the Kepler project, it was selected as a core component for this work.

2.3.5 Repository Structure

The repository for this project includes all aspects of the Kubernetes-based energy-efficiency test environment, from deployment automation to documentation. Due to the reliance on external projects, a hybrid dependency-management strategy was adopted.

2.3.5.1 Submodules for External Repositories

Several external projects that undergo frequent updates were forked and included as Git submodules. This approach allows easy customization and configuration while preserving the ability to synchronize with upstream repositories. Freezing submodules at specific commits ensures stability and avoids unexpected changes from upstream updates.

2.3.5.2 Direct Deployment from External Repositories

For external projects requiring minimal customization, direct deployment from their upstream repositories was chosen. This simplifies repository maintenance and ensures that stable, vetted versions are used.

2.3.5.3 Structure Overview

The repository is organized to maintain clarity and a clear separation of concerns:

- **ansible/:** Ansible playbooks and roles for automated deployment.
- **helm/:** Custom or external Helm charts managed through Ansible.
- **scripts/:** Bash scripts for executing Ansible playbooks.
- **config/:** Centralized configuration files and the Ansible vault.
- **docs/:** Documentation files describing setup and usage.
- **thesis/:** All thesis-related files, written in LaTeX.

2.3.6 Automation Architecture

Automation was a key focus in this project to ensure reproducibility, consistency, and ease of deployment. The automation architecture is primarily based on Ansible, with Helm commands embedded into Ansible playbooks for Kubernetes-specific deployments.

2.3.6.1 Ansible and Helm Integration

Ansible was used to automate the setup of the base environment, including system-level configurations and Kubernetes deployments. All Helm installations, such as the `kube-prometheus-stack`, were wrapped in Ansible playbooks. This approach provided a unified automation framework in which both system configurations and Kubernetes resources could be managed together. It also ensured clear version control and consistent logging of all deployment steps.

2.3.6.2 Execution Scripts

Custom Bash scripts were created to execute the Ansible playbooks. In addition to convenience, these scripts ensured:

- the correct execution context and configuration for invoking playbooks,
- automatic log creation, simplifying troubleshooting and auditing.

2.3.6.3 Centralized Configuration

All configuration values (such as IP addresses, storage paths, and deployment options) were centralized in a single configuration file. This design simplifies re-deployment on different hardware by requiring changes only in one location. When necessary, Jinja templates were used within Ansible to dynamically adapt configurations based on this central file.

2.3.6.4 Security

Sensitive information such as passwords and API keys was encrypted using an Ansible Vault. This allowed confidential data to be securely stored within the repository without compromising security during deployment.

2.4 Kepler Architecture and Metrics Collection

Because Kepler is a central component of this project, it is important to understand its architecture and how it collects metrics. This section provides a brief overview of Kepler's major components and data-collection approach. For more detailed information, the official Kepler documentation[**KeplerDocumentation**] should be consulted.

2.4.1 Kepler Components

2.4.1.1 Kepler Exporter

The core component of Kepler is the *Exporter*, which runs as a privileged DaemonSet pod on each Kubernetes node. The exporter interacts directly with the hardware and kernel to collect energy-consumption and resource-utilization metrics. It estimates power usage at the process, container, and pod levels and exposes the collected metrics in a Prometheus-compatible format.

A ServiceMonitor is also deployed, enabling Prometheus to scrape metrics from Kepler's exporter endpoints.

2.4.1.2 Kepler Model Server

Although the Kepler Model Server is not used in this project, its purpose is noteworthy. The model server provides power-estimation models at various granularities (node, pod, or component level). It may also include an online trainer that updates these models dynamically at runtime.

2.4.2 Kepler Data Collection

2.4.2.1 Process and Container Data

Kepler employs `eBPF` to collect detailed CPU event data. `eBPF` programs run in a privileged kernel context, allowing efficient, low-overhead monitoring of kernel-level events. Specifically, Kepler hooks into the `finish_task_switch` kernel function, which handles context switching, to collect process-level data. The following Perf counters are recorded:

- `PERF_COUNT_HW_CPU_CYCLES`

- `PERF_COUNT_HW_REF_CPU_CYCLES`
- `PERF_COUNT_HW_INSTRUCTIONS`
- `PERF_COUNT_HW_CACHE_MISSES`

By maintaining a BPF hash keyed by process IDs, cgroup IDs, CPU IDs, and timestamps, Kepler correlates collected events to individual processes and containers. This data forms the foundation for deriving energy-consumption estimates. The BPF hash structure is shown in Table 2.1.

TABLE 2.1: Hardware CPU events monitored by Kepler

Key	Value	Description
pid	pid	Process ID
	cgroupid	Process cgroupID
	process_run_time	Total time a process occupies the CPU (calculated each time the process leaves the CPU on a context switch)
	cpu_cycles	Total CPU cycles consumed by the process
	cpu_instr	Total CPU instructions consumed by the process
	cache_miss	Total cache misses by the process
	page_cache_hit	Total page-cache hits
	vec_nr	Total number of soft-IRQ handles (max 10)
	comm	Process name (max length 16)

2.4.2.2 CPU Power Data

Kepler leverages Intel RAPL (Running Average Power Limit) to monitor energy consumption across CPU domains such as cores, DRAM, and integrated GPUs. RAPL provides real-time power data with fine granularity, allowing Kepler to accurately measure CPU-related energy usage. The supported domains include:

- **Package (PKG):** Total energy consumption of the socket, including cores, caches, and memory controllers.
- **Power Plane 0 (PP0):** Energy consumption of CPU cores.
- **Power Plane 1 (PP1):** Energy consumption of integrated GPUs, if present.
- **DRAM:** Energy consumption of memory attached to the CPU.

To access RAPL data, Kepler uses the following methods (in order of preference):

1. **RAPL sysfs:** Direct access through the Linux power-capping framework at `/sys`. This requires root access and is the method used in this project.
2. **RAPL MSR:** Access via Model-Specific Registers, providing detailed energy readings.
3. **xgene-hwmon kernel driver:** Used on specific ARM architectures.

2.4.2.3 Platform Power Information

Kepler can also collect platform-level power data, representing the total power usage of the node. This is achieved through:

- **ACPI (Advanced Configuration and Power Interface):** Provides access to system-level power information.
- **IPMI (Intelligent Platform Management Interface):** Exposes power data via the Baseboard Management Controller (BMC).

2.4.3 Kepler Power Model

Kepler uses two complementary power-modeling approaches. If total node power is known, Kepler applies a ratio-based model to derive finer-grained power figures for individual components at the node and container levels. If detailed hardware-level readings are unavailable (for example, in virtualized environments) Kepler estimates power consumption from system-utilization metrics using a pretrained model (currently based on an Intel Xeon E5-2667 v3 processor). Because this model is processor-specific, it is inherently flawed when applied to other architectures. Increasing the number of available models is therefore a long-term goal of the project.

In previous experiments conducted by the author, Kepler was deployed on a Kubernetes cluster with virtualized nodes in an OpenStack environment. With no hardware-level power data available, Kepler attempted to estimate power consumption solely from system metrics. The resulting estimates were inconsistent and unreliable, underscoring the importance of accurate hardware data for meaningful energy analysis.

2.4.4 Metrics Produced by Kepler

Kepler collects and exports a wide range of metrics related to energy consumption and resource utilization.

2.4.4.1 Container-level Metrics

At container level, Kepler estimates total energy consumption in joules. Energy usage is broken down into the following components: cores, DRAM, uncore (such as last-level cache and memory controllers), total CPU package, GPU, and other. Additional resource-utilization metrics include total CPU time, cycles, instructions, and cache misses. Several IRQ-related metrics are also provided, such as the number of transmitted and received network packets and the number of block I/O operations.

2.4.4.2 Node-level Metrics

At node level, Kepler again estimates total energy consumption in joules. Energy estimates are provided for the whole node as well as the Core, DRAM, Uncore, CPU package, GPU, Platform, and Other categories. Kepler also exposes node-specific metadata (such as CPU architecture), aggregated metrics used by the Kepler model server, and Intel QAT utilization.

Chapter 3

Implementation

This chapter describes the implementation and configuration of the various components used in this project. All automation scripts are designed to be idempotent and are executed using shell scripts located in the `Powerstack/scripts` directory. In general, configuration is performed via the central configuration file (`Powerstack/configs/inventory.yml`), unless stated otherwise. Sensitive information is stored in the Ansible Vault file (`Powerstack/configs/vault.yml`).

3.1 K3s Installation

This section outlines the steps involved in setting up a Kubernetes cluster using K3s on bare-metal servers. Installation was automated using an Ansible playbook forked from the official `k3s-io/k3s-ansible`[122] repository, with customizations for internal IP-based communication.

3.1.1 Preparing the Nodes

Before running the Ansible playbook, the following prerequisites must be met on all servers:

- **Operating system:** Ubuntu 22.04 (kernel version 5.15.0).
- **Passwordless SSH:** A user with sudo privileges must have passwordless SSH access to each server.
- **Networking:** Each server must have both an internal IP (for cluster traffic) and an external IP (for VPN or external management).
- **Local Ansible control node setup:**
  - **ansible-community** 9.2.0 (version 8.0+ required).
  - **Python** 3.12.3 and **Jinja** 3.1.2.
  - **kubectl** 1.31.3.

3.1.2 K3s Installation with Ansible

The playbook supports x64, arm64, and armhf architectures. For this project, it was tested only on x64.

3.1.2.1 Configuration Details

- Internal and external IP addresses for all servers must be defined.
- One server must be designated as the control-plane node.
- Default values such as `ansible_user`, `ansible_port`, and the `k3s_version` may be adjusted if necessary.

3.1.2.2 Kubectl Configuration

- The playbook automatically installs and configures `kubectl` on the Ansible control node by copying the Kubernetes config file from the control-plane node.
- The user must rename the copied file from `config-new` to `config` and select the PowerStack context using:
  `kubectl config use-context powerstack`

3.2 NFS Installation and Setup

3.2.1 NFS Installation with Ansible

The NFS server and clients were fully automated via an Ansible playbook. Before beginning the automated setup, the following manual step must be completed:

- **Disk selection:** A disk must be selected on the control-plane node for persistent storage. This disk will be reformatted and all existing data will be lost.

The Ansible playbook performs the following actions:

- **Disk preparation:** The selected disk is partitioned (if required) and formatted with a single Btrfs partition occupying the full disk. The partition is mounted at `/mnt/data`, and an entry is added to `/etc/fstab` for persistence across reboots.
- **NFS server setup:** The `nfs-kernel-server` package is installed and configured on the control-plane node. The directory `/mnt/data` is exported as an NFS share for the worker nodes.
- **NFS client setup:** On each worker node, the `nfs-common` package is installed. The NFS share is mounted, and an `/etc/fstab` entry is added to ensure persistence.

3.2.1.1 Configuration Details

- The NFS network range must be specified, and all nodes must be part of that network.
- The export path must be defined.

3.3 Rancher Installation and Setup

3.3.1 Rancher Installation with Ansible and Helm

Although not strictly required for the project, Rancher was deployed in the `cattle-system` namespace to support debugging and system analysis. The installation was automated using an Ansible playbook that integrates Helm. The key steps were:

- **Helm installation:** Helm was installed on the control-plane node to deploy Rancher and its dependencies.
- **Namespace creation:** The `cattle-system` namespace was created for the Rancher deployment.
- **Cert-Manager deployment:** Cert-Manager was installed to manage TLS certificates.
- **Rancher deployment:** Rancher was installed using the official Helm chart. During installation:
  - **Hostname:** A hostname was defined for accessing Rancher.
  - The chart was configured with `-set tls=external` to enable external access.
  - **Bootstrap password:** A secure bootstrap password was set for the default administrator account.
- **Ingress configuration:** An ingress resource was created to route traffic to Rancher via the defined hostname.

3.4 Monitoring Stack Installation and Setup with Ansible

The monitoring stack (Prometheus, Grafana, and AlertManager) was deployed using the `kube-prometheus-stack`[123] Helm chart from the `prometheus-community/helm-charts` repository. Although the repository was forked for convenience, no upstream modifications were made, ensuring compatibility with future updates.

3.4.1 Prometheus and Grafana Installation with Ansible and Helm

The installation was automated using Ansible roles, ensuring idempotency and centralized configuration management. The following key steps were executed:

- **Persistent Storage Configuration:**
  - Directories for Prometheus, Grafana, and AlertManager were created on the NFS-mounted disk.
  - A custom `StorageClass` was defined for NFS storage. The default `local-path` `StorageClass` was overridden to ensure it is no longer the default.

  - PersistentVolumes (PVs) were created for Prometheus, Grafana, and AlertManager. A PersistentVolumeClaim (PVC) was created explicitly for Grafana, while the PVCs for Prometheus and AlertManager were managed by the Helm chart.

- **Helm Chart Installation:**

  - A Helm values file was generated dynamically using a Jinja template. This template incorporated variables from the central Ansible configuration file to ensure consistency. Sensitive information, such as the Grafana admin password, was included in the values file and removed from the control node after installation to mitigate security risks.

  - The Helm chart was installed via an Ansible playbook. The following customizations were applied through the generated values file:

    * PVC sizes for Prometheus and AlertManager were set based on the central configuration.

    * A Grafana admin password was defined.

    * Prometheus scrape configurations were extended to include Kepler endpoints.

    * Changes to the `securityContext` were applied to allow Prometheus to scrape Kepler metrics.

- **Service Port Forwarding:**

  - Prometheus, Grafana, and AlertManager services were exposed using static `NodePort`s defined in the central configuration file, enabling external access.

- **Cleanup:**

  - A cleanup playbook was executed to remove sensitive configuration files from both the control-plane node and the Ansible control node.

3.4.2 Removal Playbook

An Ansible playbook was created to handle complete uninstallation of the monitoring stack. This ensures that all PVs and PVCs are explicitly removed, avoiding residual artifacts in the Kubernetes cluster.

3.5 Kepler Installation and Setup with Ansible and Helm

3.5.1 Preparing the Environment

The Kepler deployment uses the official Kepler Helm chart repository. Before deploying Kepler, several prerequisites must be met to ensure correct operation.

3.5.1.1 Redfish Interface

The Redfish Scalable Platforms Management API is a RESTful API specification for out-of-band systems management. On the Lenovo servers used in this project, Redfish exposes IPMI-based power metrics, which Kepler accesses through its Redfish interface. To verify Redfish functionality, navigate to the Lenovo XClarity Controller and ensure that the following setting is enabled:

- **IPMI over LAN:** Located under `Network → Service Enablement and Port Assignment`.

Redfish API functionality can be tested using the following endpoints in a web browser:

- General Redfish information:
  `https://<BMC-IP>/redfish/v1`
- Power metrics:
  `https://<BMC-IP>/redfish/v1/Chassis/1/Power#\PowerControl`

3.5.1.2 Kernel Configuration

Kepler requires kernel-level access for `eBPF` tracing, which involves calling the `perf_event_open` syscall. By default, this syscall is restricted. To enable Kepler's tracing functionality, an Ansible role adjusts the kernel parameter `perf_event_paranoid` via `sysctl` without requiring a reboot.

The restriction level can be verified by reading `/proc/sys/kernel/perf_event_paranoid`. For this project, all restrictions were removed by setting the value to `-1`.

3.5.2 Kepler Deployment with Ansible and Helm

Kepler was deployed using the Kepler Helm chart[**Kepler_helm_chart**] from the `sustainable-computing-io/Kepler-helm-chart` repository, with Ansible automating configuration and deployment. Deployment parameters were centralized in a Jinja template, rendered locally, and copied to the control-plane node before installation.

Key configurations in the Helm values file include:

- **Enabled metrics:** Various metric sources were activated for detailed energy monitoring.
- **Service port:** The Kepler service port was defined to allow Prometheus to scrape metrics.
- **Service interval:** The Kepler service interval was set to 10 seconds.
- **Redfish metrics:** Redfish/IPMI metrics were enabled, and Redfish credentials were provided. These credentials match those used for the Lenovo XClarity Controller interface. Note that the BMC IP differs from the node's IP address.

3.5.3 Verifying Kepler Metrics

After deployment, it was essential to verify that Kepler correctly collected and exposed metrics. Verification involved the following steps:

3.5.3.1 Prometheus Scraping

After deployment, successful Prometheus scraping of the Kepler endpoints was verified via the Prometheus web interface.

3.5.3.2 Metric Availability

All Kepler metrics were inspected individually in the Prometheus web interface to ensure that non-zero values were being reported. Each metric presented a single data point, offering additional confirmation that the corresponding data source was being monitored correctly.

3.5.3.3 Kepler Logs

The Kepler logs were reviewed to examine which data sources were successfully utilized:

LISTING 3.1: `kepler.log`

```
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/online: no such file or directory
1 exporter.go:103] Kepler running on version: v0.7.12-dirty
1 config.go:293] using gCgroup ID in the BPF program: true
1 config.go:295] kernel version: 5.15
1 config.go:322] The Idle power will be exposed. Are you running on Baremetal or using single VM per node?
1 power.go:59] use sysfs to obtain power
1 node_cred.go:46] use csv file to obtain node credential
1 power.go:79] using redfish to obtain power
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/online: no such file or directory
1 exporter.go:84] Number of CPUs: 6
1 watcher.go:83] Using in cluster k8s config
1 reflector.go:351] Caches populated for *v1.Pod from pkg/kubernetes/watcher.go:211
1 watcher.go:229] k8s APIserver watcher was started
1 process_energy.go:129] Using the Ratio Power Model to estimate PROCESS_TOTAL Power
1 process_energy.go:130] Feature names: [bpf_cpu_time_ms]
1 process_energy.go:129] Using the Ratio Power Model to estimate PROCESS_COMPONENTS Power
1 process_energy.go:130] Feature names: [bpf_cpu_time_ms bpf_cpu_time_ms bpf_cpu_time_ms   gpu_compute_util]
1 node_component_energy.go:62] Skipping creation of Node Component Power Model since the system collection is
    supported
1 prometheus_collector.go:90] Registered Process Prometheus metrics
1 prometheus_collector.go:95] Registered Container Prometheus metrics
1 prometheus_collector.go:100] Registered VM Prometheus metrics
1 prometheus_collector.go:104] Registered Node Prometheus metrics
1 exporter.go:194] starting to listen on 0.0.0.0:9102
1 exporter.go:208] Started Kepler in 2.2651724s
```

3.5.3.4 Power Metrics from ACPI / IPMI

When both ACPI and IPMI were enabled for platform power measurement, Kepler preferred IPMI as its primary data source. In this case, Kepler used IPMI for overall platform energy, while relying on ACPI to derive lower-level component estimates. This behavior is expected, as IPMI typically provides more complete platform-level information. In the absence of IPMI data, Kepler automatically falls back to ACPI as the sole power source.

3.5.3.5 Redfish Issues

Kepler occasionally failed to handle individual Redfish data values correctly. These incidents were sporadic and varied across different metrics. The underlying cause could not be resolved within the scope of this thesis. The following log entry illustrates such an error:

```
Failed to get power: json: cannot unmarshal number 3.07 into Go struct
    field Voltages.Voltages.ReadingVolts of type int
```

3.5.3.6 Error Message `cpu0/online`

The following error message is noteworthy:

```
WARNING: failed to read int from file: open /sys/devices/system/cpu/cpu0/
    online: no such file or directory
```

This warning occurs because the Intel Xeon processor used in this project does not support core offlining, the dynamic disabling of individual CPU cores at runtime. While core offlining is an interesting feature for energy-efficiency analysis, this limitation can be accepted as a hardware constraint of the project.

Chapter 4

Test Procedure

This chapter describes the test procedure used to verify Kepler-produced metrics. The verification process involves executing dynamic workloads on the Kubernetes cluster and analyzing the correlation between workload intensity and the metrics reported by Kepler. For better intuitive understanding, all Joule-based metrics are converted to Watts, and all operations-based metrics are converted to IOPS.

The collected data is visualized through diagrams to support interpretation. However, this thesis does not provide a detailed energy-efficiency analysis. Instead, the goal is to verify whether Kepler metrics reliably correlate with workload fluctuations, thereby confirming its suitability for a more in-depth energy-efficiency study in future work.

4.1 Test Setup

All test workloads were created using Ansible within a dedicated Kubernetes namespace, referred to as the *testing-namespace*. While the cluster was designed to be largely hardware-independent, the test setup requires manual adjustments when deployed on different hardware. Specifically, CPU and memory allocations for test pods must be reviewed, and the storage disk used for Disk I/O experiments must be empty and correctly identified.

4.1.1 Benchmarking Pod

A dedicated Ubuntu-based *benchmarking pod* was provisioned (using Ansible) to serve as the central test agent for all experiments. This pod enabled fully self-contained testing inside the cluster, without any dependency on external machines. The benchmarking pod was configured with a complete `kubectl` setup, an OpenSSH client, and essential tools such as `wget`, `curl`, `vim`, and `git`.

4.1.2 Testing Pods

Test workloads were deployed as DaemonSets to ensure that every node in the cluster hosted the required test pods. Depending on the experiment, different resource allocations were used:

- **CPU stress testing:** A test pod and a background load pod were deployed, each with 2.5 vCPU and 1 GB memory.

- **Memory stress testing:** A test pod and a background load pod were deployed, each with 150m vCPU and 25 GB memory.
- **Network I/O and Disk I/O testing:** A single test pod was deployed with 2.5 vCPU and 20 GB memory.

Resources were allocated to ensure an even split between CPU *testing* and *background load* pods, and a similar balance for memory-intensive workloads. A resource margin was maintained to prevent system instability. Benchmarking tools were installed on each pod, including `stress-ng`[124] for CPU and memory stress tests, `fio`[125] for Disk I/O testing, and `iperf3`[126] for network performance measurements.

4.1.3 Disk Formatting and Mounting

For Disk I/O experiments, an unused HDD on each worker node was partitioned, formatted, and mounted using Ansible. Persistence across reboots was ensured through an `/etc/fstab` entry.

4.2 Test Procedure

Since energy consumption is not calculated beyond the node level, all tests were conducted on a single worker node. The test pod (either high-CPU or high-memory) generated workloads at predefined levels of 10%, 30%, 50%, 70%, and 90% for a fixed duration of 30 minutes per workload level.

For CPU and memory testing, each experiment was run under two cluster conditions:

- **Idle cluster:** No background load.
- **Busy cluster:** Background pods at 90% utilization.

Because disk and network usage are not restricted by default in Kubernetes, this distinction was not applied for Disk I/O and Network I/O tests.

4.2.1 CPU Stress Test

CPU-intensive workloads were generated using `stress-ng`, with a CPU worker initiated on each available core. This test was performed under both idle and busy cluster conditions.

4.2.2 Memory Stress Test

Memory-intensive workloads were generated using `stress-ng`, where a virtual memory worker allocated all available memory. As with the CPU tests, experiments were conducted under idle and busy cluster conditions.

4.2.3 Disk I/O Stress Test

Disk performance was evaluated using `fio`. The test consisted of two phases: first, the maximum achievable IOPS were measured using random read operations on the

mounted HDD. Next, controlled read operations were issued at predefined percentages of the measured maximum. Random reads and direct I/O were used exclusively to eliminate caching effects.

4.2.4 Network I/O Stress Test

Network performance was evaluated using `iperf3`. First, the maximum bandwidth between pods on different nodes was measured. Then, controlled tests were executed at various percentages of the maximum bandwidth. To avoid server-side measurement overhead, only client-side results were analyzed.

4.3 Data Analysis

Data collected from each experiment was analyzed in two phases using Python.

4.3.1 Data Querying

Prometheus was queried to extract Kepler metrics corresponding to each experiment's duration. The retrieved data was stored as CSV files. The analysis relied on the Python libraries `pandas`, `requests`, and `datetime` for data querying and processing.

4.3.2 Diagrams

Visualization of Kepler metric data was performed using `matplotlib`. Each diagram included:

- X-axis: Time
- Primary Y-axis: Kepler metric values (Watts or operations per second)
- Secondary Y-axis: Test workload percentage
- A moving-average overlay to improve readability

By correlating workload levels with Kepler metrics, the structured analysis validated Kepler's suitability for future energy-efficiency studies.

Chapter 5

Test Results

This chapter presents the results of the test procedures conducted to analyze Kepler-produced metrics. Each section corresponds to a specific resource type with further division into container-level and node-level metrics. The results are discussed alongside their respective figures, which illustrate Kepler-deduced energy-consumption and performance trends.

It is important to note that all Kepler metrics exhibit strong oscillations. A closer analysis shows that these oscillations follow a highly regular pattern, suggesting an issue with either Kepler's metric publication intervals or the Prometheus scraping intervals. The data has been analyzed as-is, with moving averages added to improve readability. The implications of this irregularity will be discussed further in Chapter 6.

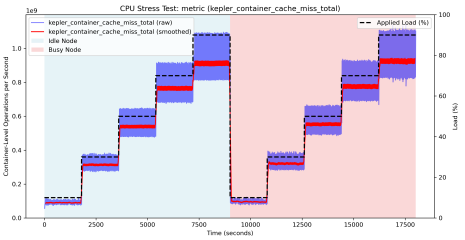
For clarity and to avoid confusion, three Kepler metric concepts are reiterated before discussing the results:

- **Package metrics:** Metrics representing the entire CPU package, including all cores and uncore components.
- **Platform metrics:** Metrics representing the entire node.
- ***Other* metrics:** Metrics capturing platform components other than the CPU package and DRAM.

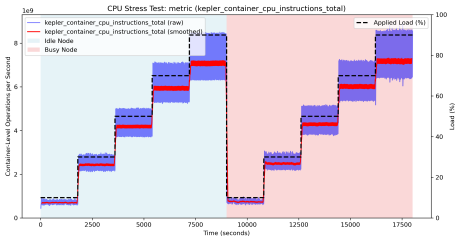
5.1 CPU Stress Test Results

5.1.1 Container-Level Metrics During a CPU Stress Test

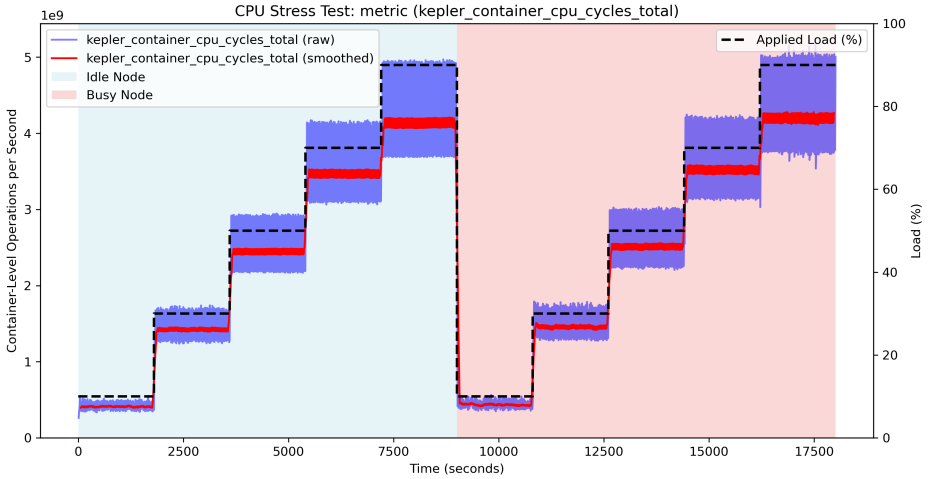
A set of figures illustrating cache misses, CPU cycles, and CPU instructions during testing is provided in Figures 5.1a to 5.1c.



(A) Cache misses



(B) CPU instructions



(C) CPU cycles

FIGURE 5.1: Kepler container-level CPU metrics during a CPU stress test

These figures illustrate cache misses, CPU cycles, and CPU instructions for the test container during execution. The diagrams show uniform trends, as the three metrics directly reflect the workload generated by `stress-ng`, which is designed to be consistent and stable. The strong correlation between the applied workload and the metrics (cache misses, CPU cycles, and CPU instructions) confirms the correct execution of the test.

Because the CPU workload running on the rest of the cluster should not affect the test container's workload, the metric values under idle and busy cluster conditions are expected to be identical. This is indeed the case, confirming that the testing procedure was executed correctly.

A figure illustrating Kepler's deduced Package energy consumption is provided in Figure 5.2.

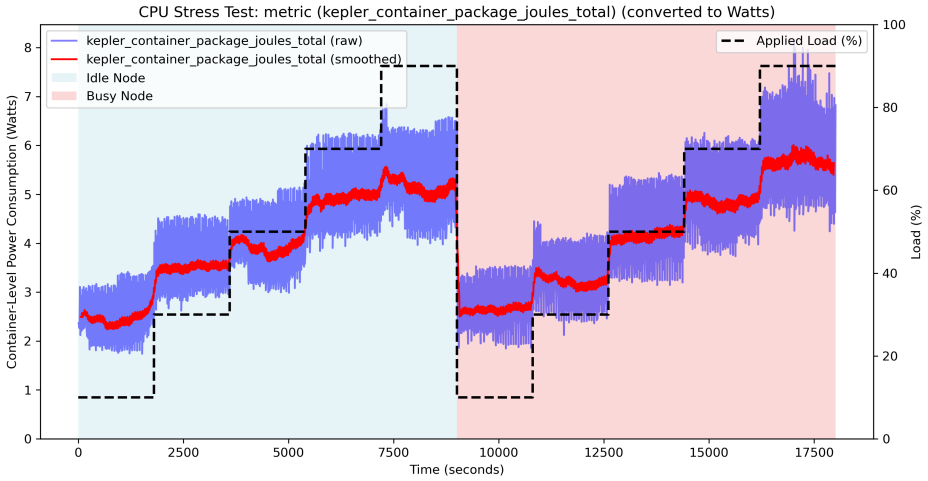
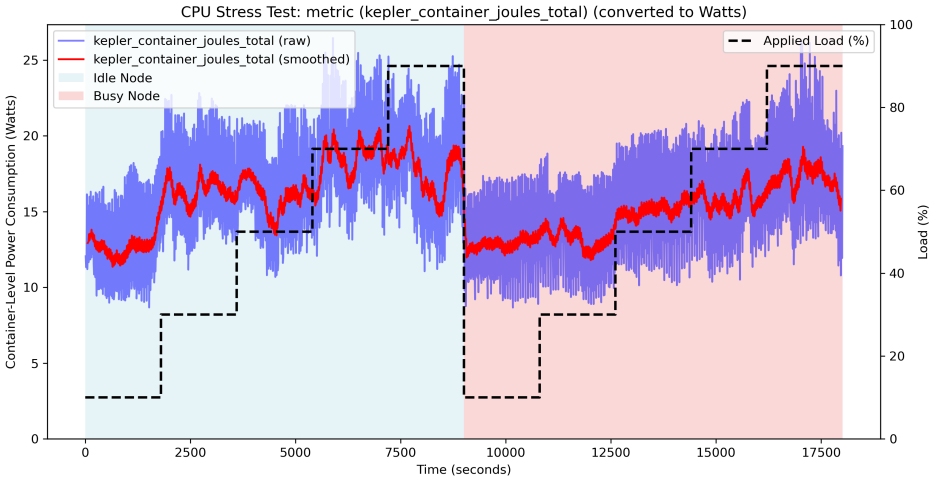


FIGURE 5.2: Kepler container-level Package energy consumption

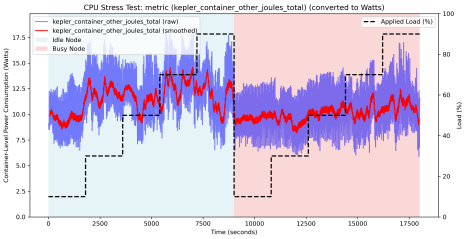
The figure shows a clear upward trend in Package energy consumption, with distinct steps that correspond to the expected workload increases. A strong correlation is observed between Kepler's reported Package energy consumption and the test workload. However, the relationship between energy consumption and workload is non-linear: while a 10% workload averages around 2.5 W, a 90% workload results in only about twice the energy consumption, despite the workload increasing by a factor of nine.

Furthermore, Kepler's Package energy measurements remain consistent regardless of whether the node is idle or busy, showing no statistically significant difference.

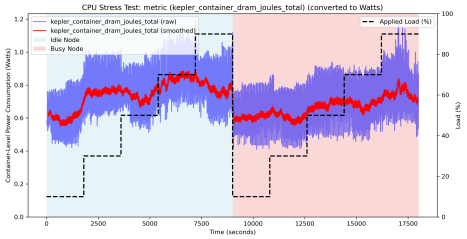
A set of figures illustrating total container energy consumption, DRAM energy consumption, and *Other* energy consumption components is provided in Figures 5.3a to 5.3c.



(A) Total container-level energy consumption



(B) *Other* energy consumption



(C) DRAM energy consumption

FIGURE 5.3: Kepler container-level energy consumption during a CPU stress test

The figures for container energy consumption, DRAM energy consumption, and *Other* components (representing host components excluding CPU and DRAM) show a less direct correlation with workload than the Package energy.

The main observations are:

- In Figure 5.3a, a slight upward trend is visible: total container energy consumption increases by roughly 5 W. This change mirrors the increase in Package energy consumption in Figure 5.2, where an approximate 5 W increase can also be observed.

- In Figure 5.3b, which shows *Other* (non-CPU/DRAM) container energy consumption, no clear trend can be identified. This is expected, since only the CPU was explicitly stressed. However, the overall magnitude of *Other* energy consumption is surprisingly high, reaching roughly twice the CPU Package energy.

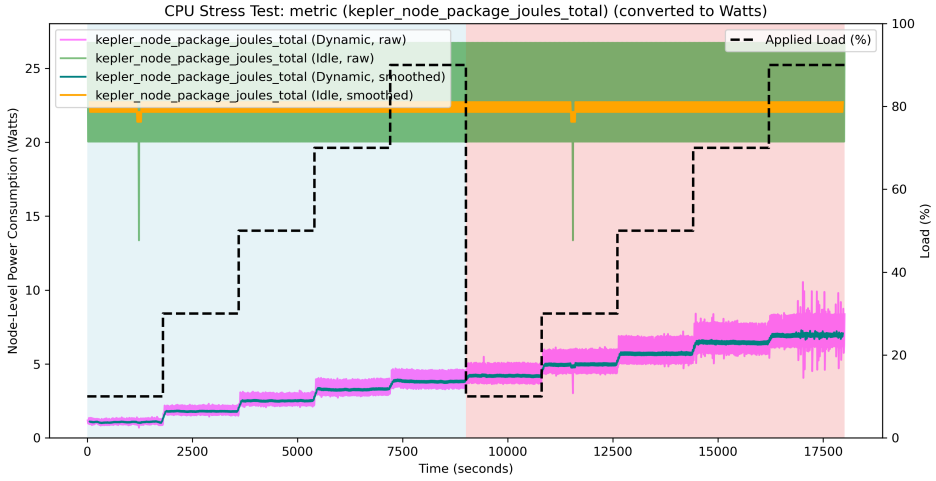
- The measured DRAM energy consumption is largely unaffected by CPU stress, as expected. With values between 0.5 and 1 W, DRAM energy remains comparatively low.

- During the second part of the experiment (the busy-node condition), all metrics appear slightly smoother, but they are neither significantly higher nor

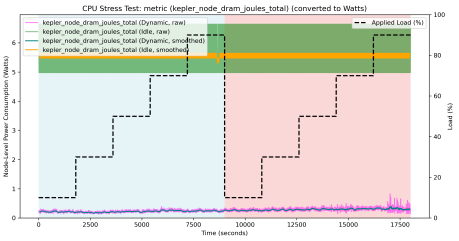
lower compared to the idle-node experiment.

5.1.2 Node-Level Metrics During a CPU Stress Test

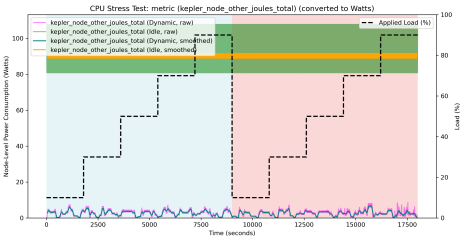
Figures illustrating node-level package, DRAM, and *Other* energy consumption are provided in Figures 5.4a to 5.4c. For node-level energy consumption, Kepler distinguishes between idle and dynamic power.



(A) Node Package energy consumption



(B) Node DRAM energy consumption



(C) Node *Other* energy consumption

FIGURE 5.4: Kepler node-level energy consumption during a CPU stress test

The following observations can be made:

- The most striking observation is the relatively high idle energy consumption of the node, which is visible in all figures. While Figure 5.4a shows increasing dynamic Package energy due to the CPU stress test, idle energy still far exceeds the dynamic component.
- The dynamic *Other* energy consumption shown in Figure 5.4c appears to be largely independent of the CPU stress load. This further supports the conclusion that *Other* system components contribute significantly to overall platform energy consumption but are generally unaffected by CPU workload.

5.1.3 Overall Conclusions

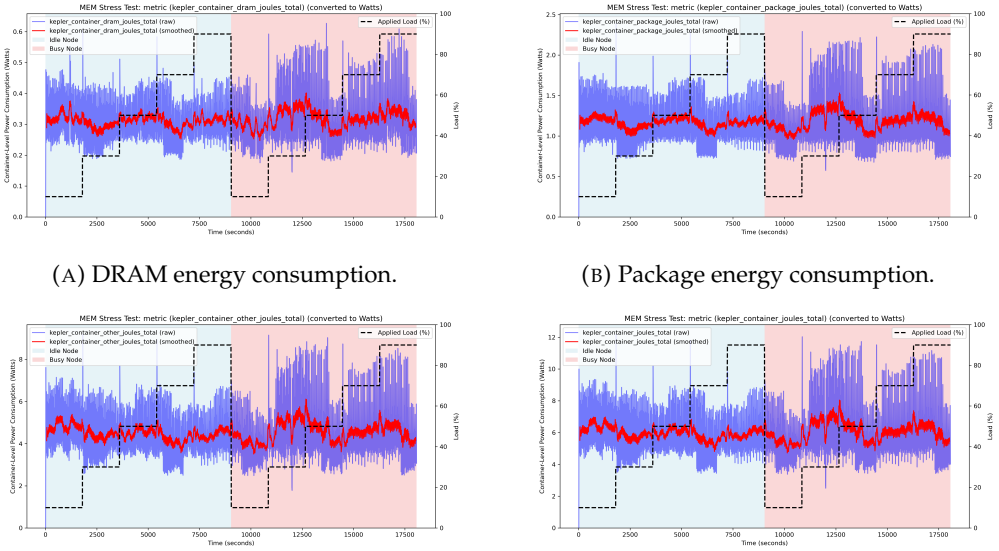
The CPU stress test results demonstrate that Kepler captures workload-dependent variations in energy consumption with reasonable accuracy. Key takeaways from the analysis include:

- Kepler's CPU Package energy measurements correlate with workload intensity, although the relationship is clearly non-linear.
- High idle energy consumption at the node level suggests that a substantial share of total energy use is independent of CPU workload.
- The *Other* component's energy consumption remains largely static with respect to workload, indicating that these components primarily contribute to baseline (idle) energy consumption rather than dynamic variations.

5.2 Memory Stress Test Results

5.2.1 Container-Level Metrics During a Memory Stress Test

Figures 5.5a to 5.5d show the container-level energy consumption metrics published by Kepler during the memory stress test.



(A) DRAM energy consumption.

(B) Package energy consumption.

(C) *Other* energy consumption.

(D) Total container energy consumption.

FIGURE 5.5: Container-level energy consumption during a memory stress test.

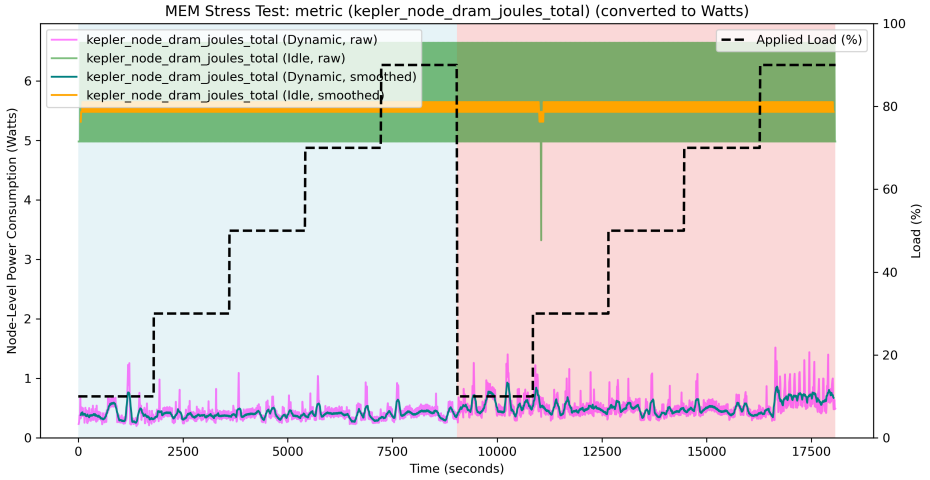
The following observations can be made:

- None of the four published energy metrics correlate with the applied memory stress load. There is also no significant difference between executing the stress test on an idle versus a busy node. None of the energy metrics indicate that a memory stress test is being performed.

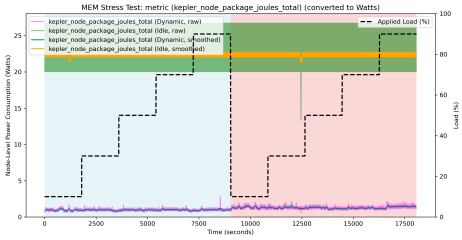
- Figure 5.5b shows an average container-level DRAM energy consumption of approximately 0.3 W. This is considerably lower than the 0.7 W measured during the CPU stress test (Figure 5.3c), which also exhibits a clear upward trend during higher CPU workloads.

5.2.2 Node-Level Metrics During a Memory Stress Test

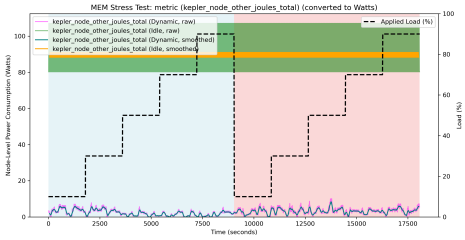
Figures 5.6a to 5.6c show the node-level idle and dynamic energy consumption metrics published by Kepler during the memory stress test.



(A) DRAM energy consumption.



(B) Package energy consumption.



(C) *Other* energy consumption.

FIGURE 5.6: Node-level energy consumption during a memory stress test.

The node-level metrics collected during the memory stress test present a similar picture to the container-level metrics. The following observations can be made:

- No node-level energy consumption metric correlates with the memory stress applied during the test.
- All node-level energy consumption metrics exhibit significantly higher idle energy consumption than dynamic energy consumption.

5.2.3 Overall Conclusions

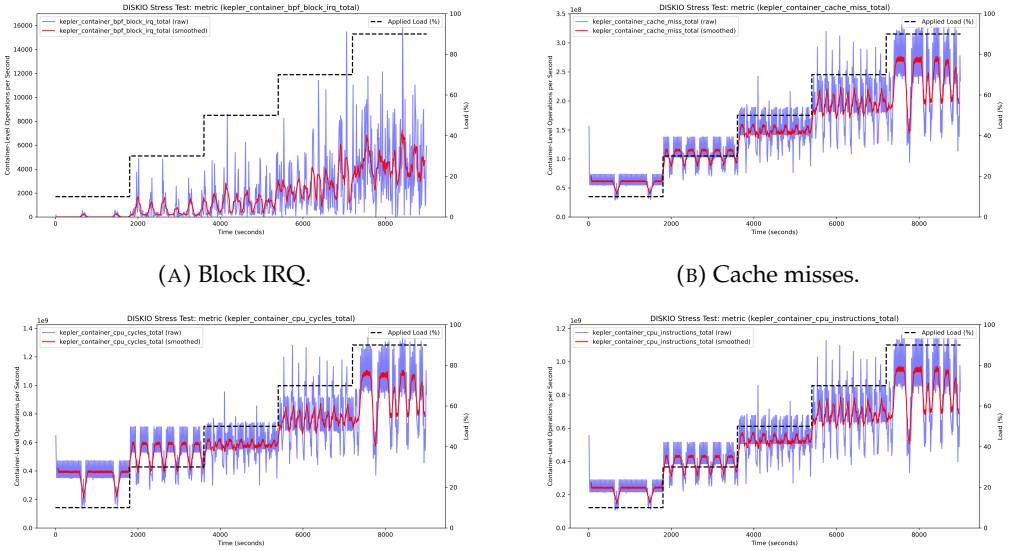
The following key takeaways can be derived from the memory stress test results:

- The memory stress test does not indicate any capability of Kepler to reliably track memory energy consumption. None of the metrics respond to the various stimuli of the test scenario.
- However, the container-level DRAM metric appears to be affected by CPU stress, as observed in the CPU stress test.

5.3 Disk I/O Stress Test Results

5.3.1 Container-Level Metrics During a Disk I/O Stress Test

Figures 5.1a to 5.7d show the CPU metrics published by Kepler during the Disk I/O stress experiment.



(A) Block IRQ.

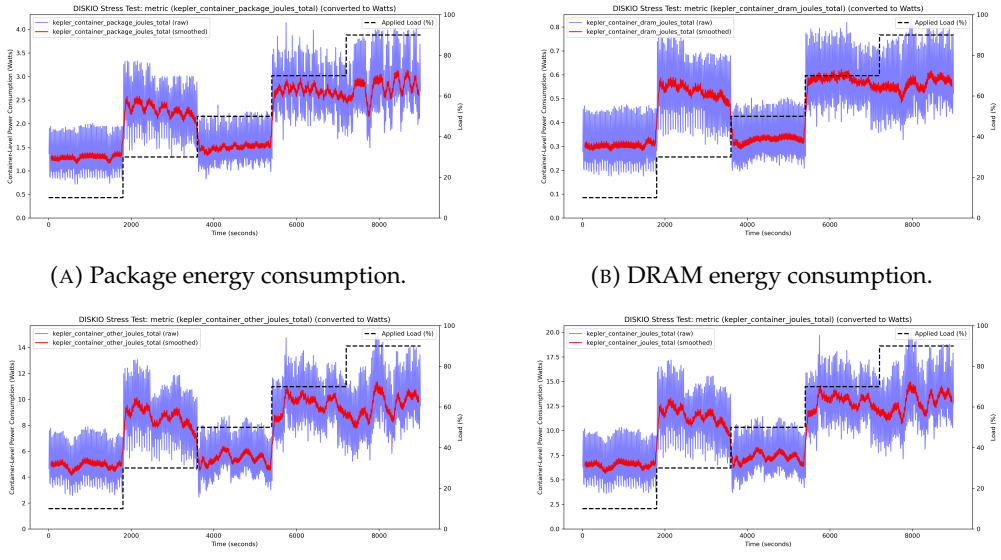
(B) Cache misses.

(C) CPU instructions.

(D) CPU cycles.

FIGURE 5.7: Container-level CPU metrics during a Disk I/O stress test.

All four figures behave as expected and validate the general test procedure. Notably, the operations per second for cache misses, CPU instructions, and CPU cycles do not scale linearly with the workload. The relative difference between the low-workload and high-workload tests is approximately 330% (cache misses), 250% (CPU instructions), and 350% (CPU cycles), all significantly lower than the 900% relative difference in the applied load.



(A) Package energy consumption.

(B) DRAM energy consumption.

(C) *Other* energy consumption.

(D) Total container energy consumption.

FIGURE 5.8: Container-level energy consumption during a Disk I/O stress test.

Figures 5.8a to 5.8d show the Kepler-reported energy metrics for the package, DRAM, *Other* components, and total container energy consumption. Several observations can be made:

- All four metrics exhibit a bimodal distribution, oscillating between distinct *low* and *high* states. While varying HDD rotation speeds might explain the curves observed for *Other* components and total container energy consumption, this does not account for the behaviour observed in package and DRAM energy consumption.
- The relative energy consumption across all four metrics appears nearly identical regardless of the component measured. This is particularly evident for the *Other* components and total container energy consumption, which appear identical aside from a scaling factor of approximately 1.5.
- All figures show strong temporal correlation with the test intervals, indicating that Kepler does detect changes in disk load.

To further analyze the metrics during a Disk I/O stress test, the experiment was repeated; the container-level Package energy consumption is shown in Figure 5.9.

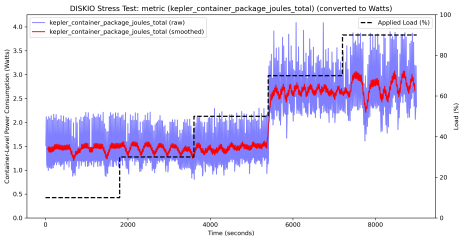


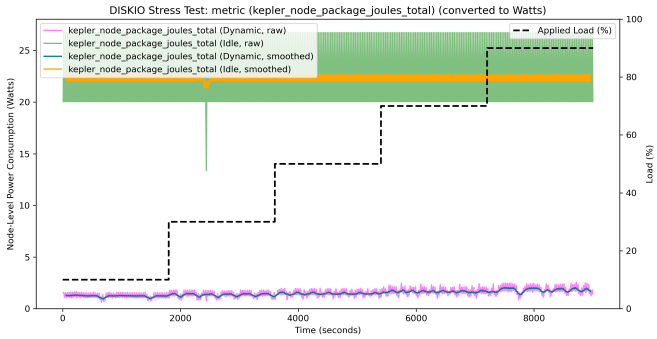
FIGURE 5.9: Container-level Package energy consumption during a second experiment.

The following observations were consistent with the first experiment:

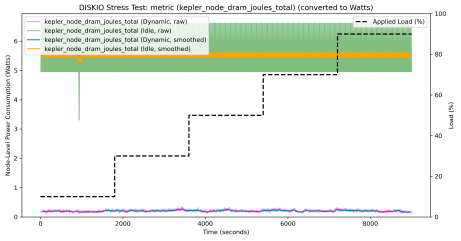
- All joule-based metrics still displayed a bimodal distribution.
- All joule-based metrics appear identical aside from differences in scale.
- All joule-based metrics show an observable upward trend.

5.3.2 Node-Level Metrics During a Disk I/O Stress Test

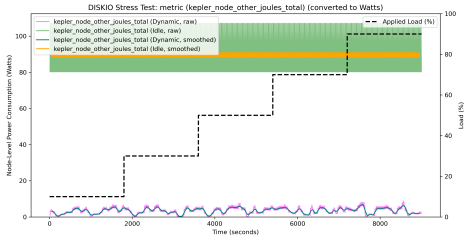
Figures 5.10a to 5.10c show the node-level energy consumption for the package, DRAM, and *Other* components, split into idle and dynamic energy consumption.



(A) Node Package energy consumption.



(B) Node DRAM energy consumption.



(C) Node *Other* energy consumption.

FIGURE 5.10: Kepler node-level energy consumption during a Disk I/O stress test.

The following observations can be made regarding node-level energy consumption:

- Across all components, the difference between idle and dynamic power is pronounced. The majority of node energy consumption occurs as idle power.
- The node DRAM and *Other* components do not show an increasing trend in dynamic energy consumption. In contrast, Figure 5.10a shows an upward trend of approximately 50% in dynamic Package energy consumption, although this remains significantly overshadowed by idle energy consumption.

5.3.3 Overall Conclusions

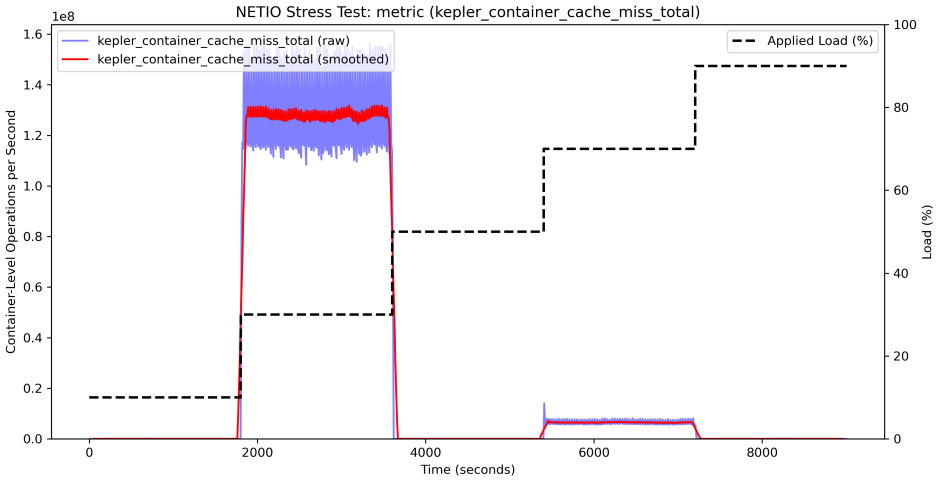
The following overall conclusions can be drawn regarding Kepler metrics during a Disk I/O test:

- Kepler provides plausible metrics for Block IRQ, cache misses, CPU instructions, and CPU cycles during a Disk I/O stress test.
- The accuracy of Kepler's container-level joule-based metrics is questionable. While they exhibit an intriguing bimodal distribution, they do not necessarily align with the Disk I/O workload.
- Kepler's container-level joule-based metrics reliably *detect* changes in Disk I/O workload, but their predictability remains uncertain.
- Node-level Kepler metrics do not suggest any significant impact of Disk I/O workload on DRAM or *Other* components.
- The fact that the variations in energy consumption shown in container-level metrics cannot clearly be traced in either idle or dynamic node energy consumption raises further questions.
- The hypothesis that Disk I/O stress does not significantly contribute to node power cannot be rejected or proven. Further research is necessary.

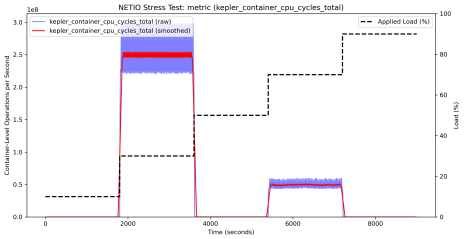
5.4 Network I/O Stress Test Results

5.4.1 Container-Level Metrics During a Network I/O Stress Test

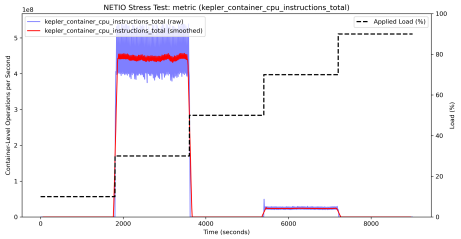
Figures 5.11a to 5.11c show the CPU metrics published by Kepler during the Network I/O stress experiment.



(A) Cache misses.



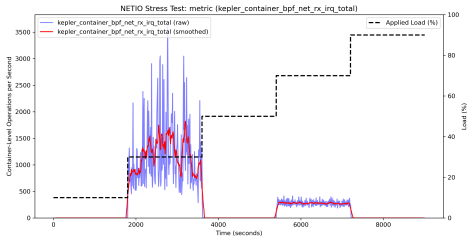
(B) CPU instructions.



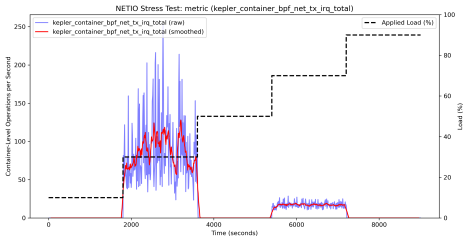
(C) CPU cycles.

FIGURE 5.11: Container-level CPU metrics during a Network I/O stress test.

The results do not provide a clear explanation for the applied test stress. While the metrics correlate in time with the applied Network I/O load, their values do not clearly correspond to the applied stress. The Kepler metrics for RX IRQ and TX IRQ in Figures 5.12a and 5.12b present a similar trend, though with significantly higher variance.

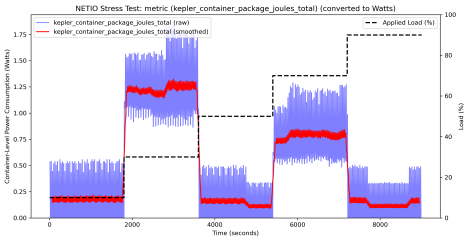


(A) RX IRQ.

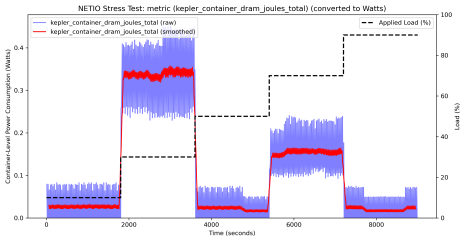


(B) TX IRQ.

FIGURE 5.12: Container-level IRQ metrics during a Network I/O stress test.

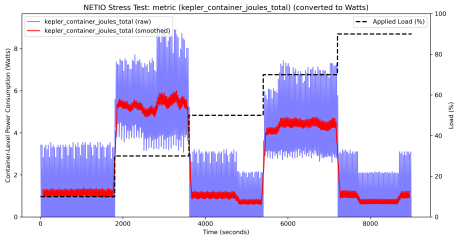


(A) Package energy consumption.



(B) DRAM energy consumption.

(C) *Other* energy consumption.



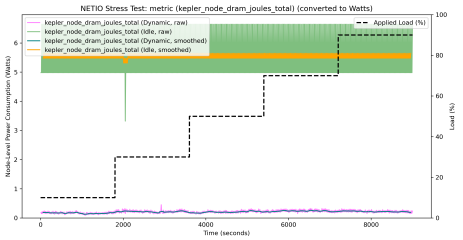
(D) Total container energy consumption.

FIGURE 5.13: Container-level energy consumption during a Network I/O stress test.

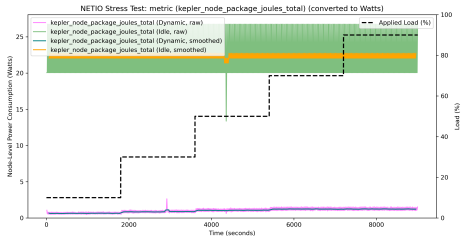
Figures 5.13a to 5.13d display Kepler metrics for package, DRAM, *Other*, and overall container-level energy consumption. The patterns remain consistent: the metrics indicate transitions between different workloads, but the values (while nearly constant for each experiment) appear unrelated to the Network I/O activity.

5.4.2 Node-Level Metrics During a Network I/O Stress Test

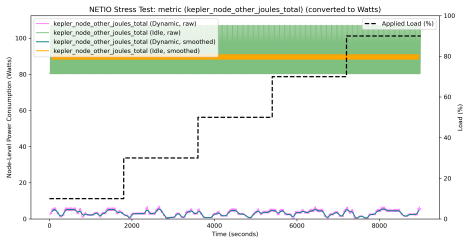
Figures 5.14a to 5.14c show the node-level idle and dynamic energy consumption metrics published by Kepler during the Network I/O stress test.



(A) DRAM energy consumption.



(B) Package energy consumption.



(C) *Other* energy consumption.

FIGURE 5.14: Node-level energy consumption during a Network I/O stress test.

Much like the previous Disk I/O test, the node-level energy consumption metrics do not show any clear indication of the applied network load. The only notable observation is the stepwise increase in dynamic Package energy consumption, which is likely related to the actual generation of network traffic.

5.4.3 Overall Conclusions

Unfortunately, the test does not indicate a clear relationship between the applied Network I/O load and the Kepler metrics. Similar to the Disk I/O test, the following conclusions can be drawn:

- The Kepler metrics are affected by the Network I/O stress test, as evidenced by their temporal correlation with the applied workload.
- However, the metric values do not logically correlate with the actual network traffic being generated.

This test was repeated multiple times with consistent results: the workload continued to correlate in time with the test steps, but the metric values plateaued at seemingly arbitrary levels for the duration of each load phase.

Chapter 6

Discussion

6.1 Conclusion and Evaluation

6.1.1 Evaluation of Cluster Setup

The cluster setup was successful and proved viable for further Kubernetes testing. While implementing the entire setup in an automated manner required substantial additional effort, the resulting deployment functioned reliably throughout the project. The ability to tear down and re-deploy the cluster at any desired depth (ranging from Kubernetes deployments and configurations to a complete reinstallation) was invaluable during experimentation. This ensured that any misconfigurations introduced during implementation could be fully removed, preventing residual effects from failed installations or incorrect configurations.

The automated setup was intentionally designed for portability across different hardware platforms. Although this capability was not tested within the project, it significantly increases the setup's reusability. Future work could adapt and deploy the setup with minimal adjustments, enabling researchers and engineers to quickly establish experimental Kubernetes clusters in diverse environments.

One of the primary constraints of this project was the decision not to implement a high-availability (HA) cluster. Given the project's focus on energy efficiency measurements rather than production-grade reliability, this was an appropriate trade-off. However, HA clusters are standard in large-scale production systems. Energy efficiency research in such environments could offer additional insights into how energy optimization strategies behave under real-world workload distributions.

Finally, graphical tools such as Rancher proved extremely useful during configuration and experimentation. Rancher's centralized UI provided a clear overview of cluster state, simplifying Kubernetes troubleshooting and management. While the project emphasized automation, Rancher complemented the setup by offering real-time monitoring and fast identification of configuration issues.

6.1.2 Evaluation of Monitoring Setup

The monitoring setup proved effective for energy consumption testing. Prometheus, the de facto standard for Kubernetes monitoring, ensured compatibility with a wide range of tools and provided access to extensive community support, documentation, and integrations. This was particularly advantageous for Kepler, which integrates

directly with Prometheus, resulting in a smooth deployment and data collection process.

A notable limitation of Prometheus is the overhead and granularity constraints introduced by periodic metric scraping. It is well suited for system monitoring at multi-second or minute-level intervals, but this limits its usefulness in high-resolution energy consumption analysis. While Kepler collects large amounts of data through eBPF and RAPL, the need to reduce data density for Prometheus-friendly exports leads to a loss of granularity. Thus, Prometheus excels at long-term trend analysis but is suboptimal for capturing rapid fluctuations in power consumption.

The use of NFS-based persistent storage on the Kubernetes control node proved reliable. Throughout the project (including multiple cluster redeployments) no monitoring data was lost. The NFS setup ensured persistent storage for Prometheus and Grafana, maintaining historical data even when the cluster was reset.

Although this monitoring setup is well suited for experimentation and research, deploying it in a production environment would require significant enhancements to ensure data integrity, resilience, and security. For instance, Prometheus data retention and storage configurations would need reinforcement, authentication and authorization mechanisms would require strengthening, and redundancy would be necessary to prevent data loss in the event of node failure.

6.1.3 Evaluation of Kepler

Kepler was integrated successfully using the provided Helm chart, although several configuration adjustments were required to ensure compatibility with the existing infrastructure. The project documentation, while helpful, has not yet reached full maturity, resulting in occasional challenges during setup and troubleshooting. Despite these hurdles, Kepler's underlying concept is well-founded, using suitable data sources to estimate energy consumption at both the container and node levels.

6.1.3.1 General Observations

- All metrics exhibited pronounced and consistent oscillations. Since the metrics are computed from simple counters, this indicates a synchronization issue, possibly between Kepler's metric publication intervals and Prometheus' scraping intervals. While this does not undermine the credibility of the data, resolving it would greatly improve the usability of the metrics.

- **CPU Energy Metrics:** Kepler successfully captures workload-dependent energy variations, showing a strong correlation between CPU stress levels and estimated power consumption. Since CPU load is the dominant factor in overall server energy consumption, this is a significant strength of Kepler.

- **Memory Energy Metrics:** Unlike CPU metrics, memory energy estimates did not show a clear correlation with applied workload. However, this is not necessarily a flaw in Kepler's methodology. Memory typically accounts for a small proportion of overall server energy consumption and varies far less than CPU power. Thus, the lack of a pronounced correlation is not unexpected. However, the experiment did not verify Kepler's ability to measure memory energy consumption.

- **Disk I/O and Network I/O Metrics:** Kepler's energy estimates for disk and network activity did not follow the expected trends. Although the metrics responded to workload transitions in a time-synchronized manner, the exact correlation remained unclear. In particular, disk and network energy consumption values were not proportional to the applied stress levels. This anomaly warrants further investigation, especially considering that HDD power consumption is known to depend only weakly on workload intensity.

6.1.4 Credible Takeaways from the Test Results

Kepler's package metrics appear reliable and provide plausible results. Although the absolute values were not validated within the scope of this project, the reported metrics closely matched the CPU workload trends observed during testing.

Energy consumption does not scale linearly with workload. For package power, increasing the workload from 10% to 90% resulted in only an approximately 250% increase in Kepler-estimated energy consumption. This aligns with the well-known observation that servers operate most efficiently at higher utilization levels.

Idle energy consumption was consistently estimated to be much higher than dynamic energy consumption. While older servers are known to exhibit high idle power usage, Kepler's estimate that idle energy consumption reaches roughly 90% on a CPU-stressed server seems unlikely and requires further investigation.

These findings suggest that Kepler's CPU energy estimation is robust, while inconsistencies in memory, disk, and network energy metrics highlight areas requiring additional validation. This naturally motivates future research into improving Kepler's measurement accuracy.

6.2 Future Work

6.2.1 Detailed Analysis of Kepler

While Kepler demonstrates strong capabilities in CPU power estimation, the inconsistencies in its memory, disk, and network metrics indicate areas requiring further research. A limitation of this study was its reliance on a single server platform. A meaningful next step would be to compare Kepler's energy estimates across different hardware configurations to evaluate generalizability.

6.2.2 Kepler Metrics Verification Through Elaborate Tests, Possibly Using Measuring Hardware

Kepler's metrics should be validated through more extensive and diverse test scenarios, using different stress tools and workloads. In some cases, integrating physical power measurement hardware could offer an additional validation layer. With deeper insight into Kepler's internal mechanisms, it may become possible to verify the reported metrics directly. Ultimately, ensuring that Kepler metrics accurately reflect energy consumption at both the node and container levels is essential for establishing their reliability.

6.2.3 Kubernetes Cluster Energy Efficiency Optimization

If Kepler metrics prove reliable for cluster-wide energy estimation, future research could investigate energy efficiency optimizations in Kubernetes environments. Potential areas of study include evaluating existing energy-saving techniques (e.g. carbon-aware schedulers), developing new optimization strategies, and analyzing the effects of different cluster configurations. For example, experiments could explore potential energy savings achieved by disabling high-availability features or dynamically powering servers on and off based on workload demand.

6.3 Final Conclusion

This project successfully established an experimental Kubernetes cluster with integrated energy monitoring. The results demonstrate that Kepler is a promising tool for CPU energy estimation, although further refinement is needed for other resource types. Going forward, improved metric validation, hardware comparisons, and research into cluster-wide optimization strategies will be essential to fully leverage Kepler for practical energy efficiency improvements.

This page was intentionally left blank.

Appendix C

System environment for development, build and debugging

System Environment for Development, Build, and Debugging

This chapter documents the system environment used to develop, build, and debug *Tycho*. It establishes the experimental and operational context required for reproducibility and auditability. Detailed operational instructions and scripts are maintained in the project repository and are referenced where appropriate [127].

1.1 Host Environment and Assumptions

All development and debugging activities were conducted on bare-metal systems rather than virtualized instances in order to preserve direct access to hardware telemetry interfaces, including RAPL, NVML, and BMC Redfish. Two physical servers were used with strictly separated roles. An auxiliary node hosted the Kubernetes control plane and supporting services, explicitly isolating orchestration overhead from the system under test.

The system under test was a dedicated compute node (product name `G494-ZU0-AAP1-000`, DALCO-integrated, MegaRAC SP-X BMC) equipped with an AMD EPYC 9554 64-core processor, 192 GB of DDR5 memory (12×16 GB), dual NVMe storage devices, and two GPUs (NVIDIA RTX 4000 Ada and NVIDIA T4). This node executed only workload pods and the *Tycho* exporter, ensuring that collected measurements reflect application and system behaviour rather than control-plane activity.

The node ran Ubuntu 22.04 LTS with a Linux 6.8 kernel (`6.8.0-90-generic`, `PREEMPT_DYNAMIC`) on `x86_64`. GPU support was provided by the NVIDIA proprietary driver (`580.95.05`) with CUDA 13.0. This software stack reflects a contemporary production configuration and provides access to recent kernel-level telemetry interfaces and current NVIDIA management APIs. Full root access was available and required, particularly for eBPF-based instrumentation. Kubernetes was installed directly on the servers using PowerStack [1] and served as the execution platform for all experiments. Access to the systems was provided via VPN and SSH within the university network.

1.2 Build and Deployment Toolchain

Development follows two complementary workflows. A local development path builds and runs the exporter directly on a target node to support interactive debugging. A deployment-oriented path builds container images, publishes them to a

registry, and deploys the exporter as a privileged DaemonSet using PowerStack.

1.2.1 Local Builds

The implementation language is Go, using `go version go1.25.1` on `linux/amd64`. A project-level `Makefile` orchestrates routine tasks. The target `make build` compiles the exporter to `_output/bin/<os>_<arch>/tycho`. Cross-compilation targets are provided for `linux/amd64` and `linux/arm64`.

Builds inject version metadata at link time via `LDFLAGS`, including source revision, branch, and build platform. This supports traceability when comparing binaries and container images across experiments.

1.2.2 Container Images and Continuous Integration

Container images are built using Docker Buildx with multi-architecture output for `linux/amd64` and `linux/arm64` and are published to the GitHub Container Registry under the project namespace. Build targets exist for a base image and optional variants that selectively enable additional software components when required.

Continuous integration is implemented using GitHub Actions. Builds produce deterministic images tagged with an immutable commit identifier, a timestamped development tag, and a `latest` tag for the `main` branch. Buildx caching is used to reduce build times without compromising reproducibility.

1.2.3 Versioning and Reproducibility

Development proceeds on feature branches with pull requests merged into `main`. Release images are produced automatically for commits on `main`, while development images are built for `dev` and selected feature branches.

Dependency management uses Go modules with a populated `vendor/` directory. The files `go.mod` and `go.sum` pin module versions, and `go mod vendor` materializes the full dependency tree to support offline and reproducible builds.

1.3 Debugging Environment

Interactive debugging uses the Delve debugger in headless mode with a Debug Adapter Protocol (DAP) listener. Delve was selected because it is purpose-built for Go, supports remote attach, and integrates reliably with common editors without requiring nonstandard build configuration beyond debug symbols.

Debug sessions are executed directly on a Kubernetes worker node. The exporter binary is started under Delve with a DAP listener bound to a dedicated TCP port. The developer workstation connects via an authenticated channel, typically using an SSH tunnel to forward the listener port. This keeps the debugger inaccessible from the wider network and avoids the need for additional cluster-level access controls.

To prevent measurement interference, the debug node excludes the deployed DaemonSet, ensuring that only the debug instance of the exporter is active on that host. Editor integration is minimal: the editor attaches to the forwarded DAP endpoint, enabling breakpoints, variable inspection, stepping, and log capture without

container-specific tooling. When the goal is system-level validation rather than interactive debugging, the deployment-oriented workflow is used and observation relies on logs and metrics instead of an attached debugger.

1.3.1 Limitations and Practical Constraints

Headless remote debugging introduces practical constraints. Interactive sessions depend on network reachability and SSH tunnelling, which adds minor latency. The debugged process must retain the privileges required for eBPF and hardware counter access, limiting where sessions can run in multi-tenant environments. Running multiple exporter instances on the same node would distort measurements, necessitating the exclusion of the DaemonSet during debugging. Container-based debugging is possible but less convenient due to the need to coordinate with cluster security policies. As a result, most active debugging is performed using locally built binaries running directly on the node, while container-based deployments are reserved for integration testing and evaluation runs.

1.4 Supporting Tools and Utilities

A lightweight configuration file `config.yaml` consolidates development toggles used for local runs, debugging sessions, and selective deployments. Repository scripts translate high-level configuration options into explicit command-line flags and environment variables for the exporter and auxiliary components. This avoids ad hoc modification of manifests or source code and aligns with the dual workflow described in § 1.2.

The surrounding toolchain includes Docker, `kubectl`, Helm, k3s, Rancher, Ansible, Prometheus, and Grafana. Each tool is used only where it reduces operational friction, for example Docker for image builds, `kubectl` for cluster interaction, and Prometheus and Grafana for observability.

1.5 Relevance, Scope, and Omissions

The environment described in this chapter constitutes enabling infrastructure rather than a scientific contribution. Its purpose is to make modifications to *Tycho* feasible and to support controlled evaluation, not to advance methodology in software engineering or tooling.

Documenting the environment supports reproducibility and auditability. A reader can verify that results were obtained on bare-metal systems with access to the required telemetry and can reconstruct the build pipeline from source to binary and container image. Operational detail is intentionally omitted from the main text and is provided in the repository documentation referenced at the start of this chapter.

Installation procedures, editor-specific configuration, system administration, security hardening, and multi-tenant policy are out of scope for this thesis. Where concrete commands are required for reproducibility, they are available in the repository documentation cited in § 1.

Bibliography

[1] Caspar Wackerle. *PowerStack: Automated Kubernetes Deployment for Energy Efficiency Analysis*. GitHub repository. 2025. URL: https://github.com/casparwackerle/PowerStack.

[2] Weiwei Lin et al. "A Taxonomy and Survey of Power Models and Power Modeling for Cloud Servers". In: *ACM Comput. Surv.* 53.5 (Sept. 2020), 100:1–100:41. ISSN: 0360-0300. DOI: 10.1145/3406208. (Visited on 04/20/2025).

[3] Saiqin Long et al. "A Review of Energy Efficiency Evaluation Technologies in Cloud Data Centers". In: *Energy and Buildings* 260 (Apr. 2022), p. 111848. ISSN: 0378-7788. DOI: 10.1016/j.enbuild.2022.111848. (Visited on 04/20/2025).

[4] Yewan Wang et al. "An Empirical Study of Power Characterization Approaches for Servers". In: *ENERGY 2019 - The Ninth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*. June 2019, p. 1. (Visited on 04/23/2025).

[5] Charles Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. New York, NY, USA: Association for Computing Machinery, Oct. 2012, pp. 1–13. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391236. (Visited on 11/26/2025).

[6] Muhammad Waseem et al. *Containerization in Multi-Cloud Environment: Roles, Strategies, Challenges, and Solutions for Effective Implementation*. July 2025. DOI: 10.48550/arXiv.2403.12980. arXiv: 2403.12980 [cs]. (Visited on 11/26/2025).

[7] Emiliano Casalicchio and Stefano Iannucci. "The State-of-the-Art in Container Technologies: Application, Orchestration and Security". In: *Concurrency and Computation: Practice and Experience* 32.17 (2020), e5668. ISSN: 1532-0634. DOI: 10.1002/cpe.5668. (Visited on 11/26/2025).

[8] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. "A Validation of DRAM RAPL Power Measurements". In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 455–470. DOI: 10.1145/2989081.2989088. (Visited on 05/21/2025).

[9] Steven van der Vlugt et al. *PowerSensor3: A Fast and Accurate Open Source Power Measurement Tool*. Apr. 2025. DOI: 10.48550/arXiv.2504.17883. arXiv: 2504.17883 [cs]. (Visited on 05/09/2025).

[10] UEFI Forum. *Advanced Configuration and Power Interface Specification Version 6.6*. Accessed April 2025. Sept. 2021. URL: https://uefi.org/sites/default/files/resources/ACPI_Spec_6.6.pdf.

[11] Richard Kavanagh, Django Armstrong, and Karim Djemame. "Accuracy of Energy Model Calibration with IPMI". In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. June 2016, pp. 648–655. DOI: 10.1109/CLOUD.2016.0091. (Visited on 04/23/2025).

[12] Richard Kavanagh and Karim Djemame. "Rapid and Accurate Energy Models through Calibration with IPMI and RAPL". In: *Concurrency and Computation: Practice and Experience* 31.13 (2019), e5124. ISSN: 1532-0634. DOI: 10.1002/cpe.5124. (Visited on 04/23/2025).

[13] Magnus Herrlin. "Accessing Onboard Server Sensors for Energy Efficiency in Data Centers". In: (Sept. 2021). (Visited on 11/27/2025).

[14] Ghazanfar Ali et al. "Redfish-Nagios: A Scalable Out-of-Band Data Center Monitoring Framework Based on Redfish Telemetry Model". In: *Fifth International Workshop on Systems and Network Telemetry and Analytics*. Minneapolis MN USA: ACM, June 2022, pp. 3–11. ISBN: 978-1-4503-9315-7. DOI: 10.1145/3526064.3534108. (Visited on 11/27/2025).

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 3B, Chapter 16.10: Plaform Specific Power Management Support. Available online: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. 2024.

[16] Guillaume Raffin and Denis Trystram. "Dissecting the Software-Based Measurement of CPU Energy Consumption: A Comparative Analysis". In: *IEEE Transactions on Parallel and Distributed Systems* 36.1 (Jan. 2025), pp. 96–107. ISSN: 1558-2183. DOI: 10.1109/TPDS.2024.3492336. (Visited on 04/02/2025).

[17] Daniel Hackenberg et al. "An Energy Efficiency Feature Survey of the Intel Haswell Processor". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. May 2015, pp. 896–904. DOI: 10.1109/IPDPSW.2015.70. (Visited on 04/28/2025).

[18] Daniel Hackenberg et al. "Power Measurement Techniques on Standard Compute Nodes: A Quantitative Comparison". In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2013, pp. 194–204. DOI: 10.1109/ISPASS.2013.6557170. (Visited on 04/28/2025).

[19] Lukas Alt et al. "An Experimental Setup to Evaluate RAPL Energy Counters for Heterogeneous Memory". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24. New York, NY, USA: Association for Computing Machinery, May 2024, pp. 71–82. ISBN: 979-8-4007-0444-4. DOI: 10.1145/3629526.3645052. (Visited on 04/02/2025).

[20] Tom Kennes. *Measuring IT Carbon Footprint: What Is the Current Status Actually?* June 2023. DOI: 10.48550/arXiv.2306.10049. arXiv: 2306.10049 [cs]. (Visited on 04/23/2025).

[21] Robert Schöne et al. "Energy Efficiency Features of the Intel Alder Lake Architecture". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. London United Kingdom: ACM, May 2024, pp. 95–106. ISBN: 979-8-4007-0444-4. DOI: 10.1145/3629526.3645040. (Visited on 04/07/2025).

[22] Robert Schöne et al. "Energy Efficiency Aspects of the AMD Zen 2 Architecture". In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2021, pp. 562–571. DOI: 10.1109/Cluster48925.2021.00087. (Visited on 04/28/2025).

[23] Kashif Nizam Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (Mar. 2018), 9:1–9:26. ISSN: 2376-3639. DOI: 10.1145/3177754. (Visited on 04/07/2025).

[24] Mathilde Jay et al. "An Experimental Comparison of Software-Based Power Meters: Focus on CPU and GPU". In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. May 2023, pp. 106–118. DOI: 10.1109/CCGrid57682.2023.00020. (Visited on 04/21/2025).

[25] Moritz Lipp et al. "PLATYPUS: Software-based Power Side-Channel Attacks on X86". In: *2021 IEEE Symposium on Security and Privacy (SP)*. May 2021, pp. 355–371. DOI: 10.1109/SP40001.2021.00063. (Visited on 05/21/2025).

[26] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers*. Tech. rep. 335592-081US. Accessed 2025-04-28. Intel Corporation, Sept. 2023. URL: https://cdrdv2.intel.com/v1/dl/getContent/671098.

[27] Zeyu Yang, Karel Adamek, and Wesley Armour. "Accurate and Convenient Energy Measurements for GPUs: A Detailed Study of NVIDIA GPU's Built-In Power Sensor". In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2024, pp. 1–17. DOI: 10.1109/SC41406.2024.00028. (Visited on 05/09/2025).

[28] Oscar Hernandez et al. "Preliminary Study on Fine-Grained Power and Energy Measurements on Grace Hopper GH200 with Open-Source Performance Tools". In: *Proceedings of the 2025 International Conference on High Performance Computing in Asia-Pacific Region Workshops*. Hsinchu Taiwan: ACM, Feb. 2025, pp. 11–22. ISBN: 979-8-4007-1342-2. DOI: 10.1145/3703001.3724383. (Visited on 11/27/2025).

[29] Le Mai Weakley et al. "Monitoring and Characterizing GPU Usage". In: *Concurrency and Computation: Practice and Experience* 37.3 (2025), e8341. ISSN: 1532-0634. DOI: 10.1002/cpe.8341. (Visited on 11/27/2025).

[30] The Linux Kernel Community. *The proc Filesystem*. https://www.kernel.org/doc/html/latest/filesystems/proc.html. Accessed: 2025-06-17. 2025.

[31] The Linux Kernel Community. *Control Group v1 — Linux Kernel Documentation*. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/index.html. Accessed: 2025-06-17. 2025.

[32] The Linux Kernel Community. *Control Group v2 — Linux Kernel Documentation*. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html. Accessed: 2025-06-17. 2025.

[33] Cilium Authors. *eBPF and XDP Reference Guide*. https://docs.cilium.io/en/latest/reference-guides/bpf/index.html. Accessed: 2025-06-17. 2025.

[34] Cyril Cassagnes et al. "The Rise of eBPF for Non-Intrusive Performance Monitoring". In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. Apr. 2020, pp. 1–7. DOI: 10.1109/NOMS47738.2020.9110434. (Visited on 06/14/2025).

[35] Brendan Gregg. *CPU Utilization is Wrong*. Blog post. Accessed 29 June 2025. May 2017. URL: https://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html.

[36] smartmontools developers. *smartmontools: Control and monitor storage systems using S.M.A.R.T.* https://github.com/smartmontools/smartmontools/. Accessed May 2025. 2025.

[37] Linux NVMe Maintainers. *nvme-cli: NVMe management command line interface*. https://github.com/linux-nvme/nvme-cli. Accessed May 2025. 2025.

[38] Seokhei Cho et al. "Design Tradeoffs of SSDs: From Energy Consumption's Perspective". In: *ACM Trans. Storage* 11.2 (Mar. 2015), 8:1–8:24. ISSN: 1553-3077. DOI: 10.1145/2644818. (Visited on 05/18/2025).

[39] Yan Li and Darrell D.E. Long. "Which Storage Device Is the Greenest? Modeling the Energy Cost of I/O Workloads". In: *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*. Sept. 2014, pp. 100–105. DOI: 10.1109/MASCOTS.2014.20. (Visited on 05/19/2025).

[40] Eric Borba, Eduardo Tavares, and Paulo Maciel. "A Modeling Approach for Estimating Performance and Energy Consumption of Storage Systems". In: *Journal of Computer and System Sciences* 128 (Sept. 2022), pp. 86–106. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2022.04.001. (Visited on 05/18/2025).

[41] Ripduman Sohan et al. "Characterizing 10 Gbps Network Interface Energy Consumption". In: *IEEE Local Computer Network Conference*. Oct. 2010, pp. 268–271. DOI: 10.1109/LCN.2010.5735719. (Visited on 05/30/2025).

[42] Robert Basmadjian et al. "Cloud Computing and Its Interest in Saving Energy: The Use Case of a Private Cloud". In: *Journal of Cloud Computing: Advances, Systems and Applications* 1.1 (June 2012), p. 5. ISSN: 2192-113X. DOI: 10.1186/2192-113X-1-5. (Visited on 06/01/2025).

[43] Saeedeh Baneshi et al. "Analyzing Per-Application Energy Consumption in a Multi-Application Computing Continuum". In: *2024 9th International Conference on Fog and Mobile Edge Computing (FMEC)*. Sept. 2024, pp. 30–37. DOI: 10.1109/FMEC62297.2024.10710253. (Visited on 05/30/2025).

[44] TechNotes. *Deciphering the PCI Power States*. Accessed June 2025. Feb. 2024. URL: https://technotes.blog/2024/02/04/deciphering-the-pci-power-states/.

[45] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. "Power Provisioning for a Warehouse-Sized Computer". In: *SIGARCH Comput. Archit. News* 35.2 (June 2007), pp. 13–23. ISSN: 0163-5964. DOI: 10.1145/1273440.1250665. (Visited on 05/21/2025).

[46] Chung-Hsing Hsu and Stephen W. Poole. "Power Signature Analysis of the SPECpower_ssj2008 Benchmark". In: *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. Apr. 2011, pp. 227–236. DOI: 10.1109/ISPASS.2011.5762739. (Visited on 05/21/2025).

[47] Shuaiwen Leon Song, Kevin Barker, and Darren Kerbyson. "Unified Performance and Power Modeling of Scientific Workloads". In: *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*. E2SC '13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 1–8. ISBN: 978-1-4503-2504-2. DOI: 10.1145/2536430.2536435. (Visited on 05/21/2025).

[48] Jordi Arjona Aroca et al. "A Measurement-Based Analysis of the Energy Consumption of Data Center Servers". In: *Proceedings of the 5th International Conference on Future Energy Systems*. E-Energy '14. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 63–74. ISBN: 978-1-4503-2819-7. DOI: 10.1145/2602044.2602061. (Visited on 06/01/2025).

[49] Inc. Meta Platforms. *Kepler v0.9.0 (pre-rewrite): Kubernetes-based power and energy estimation framework*. Accessed: 2025-04-28. 2023. URL: https://https://github.com/sustainable-computing-io/kepler/releases/tag/v0.9.0.

[50] Bjorn Pijnacker. "Estimating Container-level Power Usage in Kubernetes". MA thesis. University of Groningen, Nov. 2024. (Visited on 03/17/2025).

[51] Linux Foundation Energy and Performance Working Group. *Kepler: Kubernetes-based Power and Energy Estimation Framework*. Accessed: 2025-11-14. 2025. URL: https://github.com/sustainable-computing-io/kepler.

[52] Bjorn Pijnacker, Brian Setz, and Vasilios Andrikopoulos. *Container-Level Energy Observability in Kubernetes Clusters*. Apr. 2025. DOI: 10.48550/arXiv.2504.10702. arXiv: 2504.10702 [cs]. (Visited on 07/02/2025).

[53] Hubblo-org. *Scaphandre Documentation*. Accessed: 2025-04-28. 2024. URL: https://github.com/hubblo-org/scaphandre-documentation.

[54] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. "SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. May 2020, pp. 479–488. DOI: 10.1109/CCGrid49817.2020.00-45. (Visited on 05/21/2025).

[55] MLCO2. *CodeCarbon: Track emissions from your computing*. Accessed: 2025-04-28. 2023. URL: https://github.com/mlco2/codecarbon.

[56] International Energy Agency. *Energy and AI*. Licence: CC BY 4.0. Paris, 2025. URL: https://www.iea.org/reports/energy-and-ai.

[57] Ryan Smith. *Intel's CEO Says Moore's Law Is Slowing to a Three-Year Cadence — But It's Not Dead Yet*. Accessed: 2025-04-14. 2023. URL: https://www.tomshardware.com/tech-industry/semiconductors/intels-ceo-says-moores-law-is-slowing-to-a-three-year-cadence-but-its-not-dead-yet.

[58] Martin Keegan. *The End of Dennard Scaling*. Accessed: 2025-04-14. 2013. URL: https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/.

[59] Uptime Institute. *Global PUEs – Are They Going Anywhere?* Accessed: 2025-04-14. 2023. URL: https://journal.uptimeinstitute.com/global-pues-are-they-going-anywhere/.

[60] Eric Masanet et al. "Recalibrating global data center energy-use estimates". In: *Science* 367.6481 (2020), pp. 984–986. DOI: 10.1126/science.aba3758. eprint: https://www.science.org/doi/pdf/10.1126/science.aba3758. URL: https://www.science.org/doi/abs/10.1126/science.aba3758.

[61] Amit M. Potdar et al. "Performance Evaluation of Docker Container and Virtual Machine". In: *Procedia Computer Science* 171 (2020), pp. 1419–1428. ISSN: 1877-0509. DOI: 10.1016/j.procs.2020.04.152.

[62] Roberto Morabito. "Power Consumption of Virtualization Technologies: An Empirical Investigation". In: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. Dec. 2015, pp. 522–527. DOI: 10.1109/UCC.2015.93. (Visited on 05/21/2025).

[63] OpenAI. *ChatGPT (Version 4o)*. Used for document generation and formatting. 2025. URL: https://chat.openai.com.

[64] Chaoqiang Jin et al. "A Review of Power Consumption Models of Servers in Data Centers". In: *Applied Energy* 265 (May 2020), p. 114806. ISSN: 0306-2619. DOI: 10.1016/j.apenergy.2020.114806. (Visited on 03/16/2025).

[65] Hannes Tröpgen et al. "16 Years of SPEC Power: An Analysis of X86 Energy Efficiency Trends". In: *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*. Sept. 2024, pp. 76–80. DOI: 10.1109/CLUSTERWorkshops61563.2024.00020. (Visited on 04/20/2025).

[66] Joseph P. White et al. "Monitoring and Analysis of Power Consumption on HPC Clusters Using XDMoD". In: *Practice and Experience in Advanced Research Computing*. Portland OR USA: ACM, July 2020, pp. 112–119. ISBN: 978-1-4503-6689-2. DOI: 10.1145/3311790.3396624. (Visited on 04/23/2025).

[67] Thomas-Krenn.AG. *Redfish - Thomas-Krenn-Wiki*. Accessed: April 27, 2025. n.d. URL: https://www.thomas-krenn.com/de/wiki/Redfish.

[68] Project Exigence. *Running Average Power Limit (RAPL)*. https://projectexigence.eu/green-ict-digest/running-average-power-limit-rapl/. Accessed April 2025. n.d.

[69] AMD. *amd_energy: AMD Energy Driver*. Accessed: 2025-04-28. 2023. URL: https://github.com/amd/amd_energy.

[70] Marcus Hähnel et al. "Measuring Energy Consumption for Short Code Paths Using RAPL". In: *SIGMETRICS Perform. Eval. Rev.* 40.3 (Jan. 2012), pp. 13–17. ISSN: 0163-5999. DOI: 10.1145/2425248.2425252. (Visited on 05/21/2025).

[71] "Detailed and Simultaneous Power and Performance Analysis - Servat - 2016 - Concurrency and Computation: Practice and Experience - Wiley Online Library". In: (). (Visited on 05/21/2025).

[72] Green Coding Berlin. *RAPL, SGX and energy filtering - Influences on power consumption*. Accessed May 2025. 2022. URL: https://www.green-coding.io/case-studies/rapl-and-sgx/.

[73] Intel Corporation. *Running Average Power Limit (RAPL) Energy Reporting*. Accessed May 2025. 2022. URL: https://www.intel.cn/content/www/cn/zh/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[74] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. New York, NY, USA: Association for Computing Machinery, June 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246. (Visited on 05/21/2025).

[75] Kubernetes Documentation. *GPUs in Kubernetes*. Accessed: 2025-05-09. 2025. URL: https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/.

[76] Inc. Datadog. *2024 Container Report*. Accessed: 2025-05-09. 2024. URL: https://www.datadoghq.com/container-report/.

[77] TensorFlow Documentation. *Running TensorFlow on Kubernetes*. Accessed: 2025-05-09. 2025. URL: https://www.tensorflow.org/tfx/serving/serving_kubernetes.

[78] NVIDIA Corporation. *NVIDIA Virtualization Resources*. Accessed: 2025-05-09. 2025. URL: https://www.nvidia.com/de-de/data-center/virtualization/resources/.

[79] AMD Corporation. *AMD Instinct Virtualization Documentation*. Accessed: 2025-05-09. 2025. URL: https://instinct.docs.amd.com/projects/virt-drv/en/latest/index.html.

[80] NVIDIA Corporation. *NVIDIA Multi-Instance GPU (MIG) User Guide*. Accessed: 2025-05-09. 2025. URL: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[81] NVIDIA Corporation. *NVIDIA GPU Passthrough Documentation*. Accessed: 2025-05-09. 2025. URL: https://docs.nvidia.com/datacenter/tesla/gpu-passthrough/index.html.

[82] NVIDIA Corporation. *NVIDIA System Management Interface (nvidia-smi)*. Accessed: 2025-05-09. 2025. URL: https://developer.nvidia.com/system-management-interface.

[83] Vijay Kandiah et al. "AccelWattch: A Power Modeling Framework for Modern GPUs". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 738–753. ISBN: 978-1-4503-8557-2. DOI: 10.1145/3466752.3480063. (Visited on 05/09/2025).

[84] Varsha Singhania, Shaizeen Aga, and Mohamed Assem Ibrahim. *FinGraV: Methodology for Fine-Grain GPU Power Visibility and Insights*. Mar. 2025. DOI: 10.48550/arXiv.2412.12426. arXiv: 2412.12426 [cs]. (Visited on 05/09/2025).

[85] Anthony Hylick et al. "An Analysis of Hard Drive Energy Consumption". In: *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*. Sept. 2008, pp. 1–10. DOI: 10.1109/MASCOT.2008.4770567. (Visited on 05/21/2025).

[86] Anton Beloglazov et al. "Chapter 3 - A Taxonomy and Survey of Energy-Efficient Data Centers and Cloud Computing Systems". In: *Advances in Computers*. Ed. by Marvin V. Zelkowitz. Vol. 82. Elsevier, Jan. 2011, pp. 47–111. DOI: 10.1016/B978-0-12-385512-1.00003-7. (Visited on 06/09/2025).

[87] E. N. (Mootaz) Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. "Energy-Efficient Server Clusters". In: *Power-Aware Computer Systems*. Ed. by Babak Falsafi and T. N. Vijaykumar. Berlin, Heidelberg: Springer, 2003, pp. 179–197. ISBN: 978-3-540-36612-6. DOI: 10.1007/3-540-36612-1_12.

[88] Avita Katal, Susheela Dahiya, and Tanupriya Choudhury. "Energy Efficiency in Cloud Computing Data Center: A Survey on Hardware Technologies". In: *Cluster Computing* 25.1 (Feb. 2022), pp. 675–705. ISSN: 1573-7543. DOI: 10.1007/s10586-021-03431-z. (Visited on 06/04/2025).

[89] Robert Basmadjian et al. "A Methodology to Predict the Power Consumption of Servers in Data Centres". In: *Proceedings of the 2nd International Conference on Energy-Efficient Computing and Networking*. E-Energy '11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 1–10. ISBN: 978-1-4503-1313-1. DOI: 10.1145/2318716.2318718. (Visited on 06/09/2025).

[90] L. Luo, W.-J Wu, and F. Zhang. "Energy modeling based on cloud data center". In: *Ruan Jian Xue Bao/Journal of Software* 25 (July 2014), pp. 1371–1387. DOI: 10.13328/j.cnki.jos.004604.

[91] Robert Basmadjian and Hermann De Meer. "Evaluating and modeling power consumption of multi-core processors". In: *Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet*. 2012, pp. 1–10.

[92] Osman Sarood et al. "Maximizing throughput of overprovisioned hpc data centers under a strict power budget". In: *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 807–818.

[93] Weiwei Lin et al. "A cloud server energy consumption measurement system for heterogeneous cloud environments". In: *Information Sciences* 468 (2018), pp. 47–62.

[94] Patricia Arroba et al. "Server power modeling for run-time energy optimization of cloud computing facilities". In: *Energy Procedia* 62 (2014), pp. 401–410.

[95] Aman Kansal et al. "Virtual machine power metering and provisioning". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 39–50.

[96] Miriam Allalouf et al. "Storage Modeling for Power Estimation". In: *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. SYSTOR '09. New York, NY, USA: Association for Computing Machinery, May 2009, pp. 1–10. ISBN: 978-1-60558-623-6. DOI: 10.1145/1534530.1534535. (Visited on 05/18/2025).

[97] StoreDbits. *Hard Drive Power Consumption (HDD)*. Accessed May 2025. 2023. URL: https://storedbits.com/hard-drive-power-consumption/.

[98] StoreDbits. *SSD Power Consumption*. Accessed May 2025. 2023. URL: https://storedbits.com/ssd-power-consumption/.

[99] Corey Gough, Ian Steiner, and Winston A. Sanders. *Energy Efficient Servers: Blueprints for Data Center Optimization*. Apress, 2015. ISBN: 9781430266372. DOI: 10.1007/978-1-4302-6638-9.

[100] Vincenzo De Maio et al. "Modelling Energy Consumption of Network Transfers and Virtual Machine Migration". In: *Future Generation Computer Systems* 56 (Mar. 2016), pp. 388–406. ISSN: 0167-739X. DOI: 10.1016/j.future.2015.07.007. (Visited on 06/04/2025).

[101] Waltenegus Dargie and Jianjun Wen. "A Probabilistic Model for Estimating the Power Consumption of Processors and Network Interface Cards". In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. July 2013, pp. 845–852. DOI: 10.1109/TrustCom.2013.103. (Visited on 06/04/2025).

[102] Google Inc. *cAdvisor: Analyzes Resource Usage and Performance Characteristics of Running Containers*. https://github.com/google/cadvisor. Accessed: 2025-06-14. 2025.

[103] Kubernetes-SIG. *metrics-server: Scalable and efficient source of container resource metrics for Kubernetes built-in autoscaling pipelines*. https://github.com/kubernetes-sigs/metrics-server. Accessed: 2025-06-17. 2025.

[104] Arne Tarara. *CPU Utilization – A Useful Metric?* Green Coding Case Study. Accessed 29 June 2025. June 2023. URL: https://www.green-coding.io/case-studies/cpu-utilization-usefulness/.

[105] Adrian Cockcroft. "Utilization is virtually useless as a metric!" In: *Int. CMG Conference*. 2006, pp. 557–562.

[106] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. New York, NY, USA: Cambridge University Press, 2013. ISBN: 9781107027503.

[107] CodeCarbon Team. *mlco2/codecarbon: v2.4.1*. Version v2.4.1. May 2024. DOI: 10.5281/zenodo.11171501. URL: https://doi.org/10.5281/zenodo.11171501.

[108] CodeCarbon Contributors. *CodeCarbon issue #322: API endpoint and swagger docs*. https://github.com/mlco2/codecarbon/issues/322. Accessed: 2025-06-21. 2022.

[109] GreenAI-UPPA. *AI PowerMeter: A Tool to Estimate the Energy Consumption of AI Workloads*. Accessed: 2025-04-28. 2023. URL: https://greenai-uppa.github.io/AIPowerMeter/.

[110] Pierre Fieni et al. *PowerAPI is a green-computing toolbox to measure, analyze, and optimize the energy consumption of the various hardware/software levels composing an infrastructure*. https://github.com/powerapi-ng. Accessed June 2025. 2024.

[111] Guillaume Fieni et al. "PowerAPI: A Python Framework for Building Software-Defined Power Meters". In: *Journal of Open Source Software* 9.98 (June 2024), p. 6670. DOI: 10.21105/joss.06670. (Visited on 06/21/2025).

[112] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[113] Arne Tarara. *Green Coding Documentation*. Accessed: 2025-04-28. 2023. URL: https://github.com/green-coding-solutions/green-metrics-tool.

[114] Marcelo Amaral et al. "Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications". In: *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. July 2023, pp. 69–71. DOI: 10.1109/CLOUD60044.2023.00017. (Visited on 03/10/2025).

[115] Sustainable Computing. *Kepler: Kubernetes Efficient Power Level Exporter Documentation (Deprecated)*. https://sustainable-computing.io/. Deprecated documentation, Accessed: June. 2025.

[116] Sunyanan Choochotkaew et al. "Advancing Cloud Sustainability: A Versatile Framework for Container Power Model Training". In: *2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Oct. 2023, pp. 1–4. DOI: 10.1109/MASCOTS59514.2023.10387542. (Visited on 07/02/2025).

[117] Marcelo Amaral et al. *Exploring Kepler's Potentials: Unveiling Cloud Application Power Consumption*. https://www.cncf.io/blog/2023/10/11/exploring-keplers-potentials-unveiling-cloud-application-power-consumption/. CNCF Blog. Oct. 2023. URL: https://www.cncf.io/blog/2023/10/11/exploring-keplers-potentials-unveiling-cloud-application-power-consumption/.

[118] Global e-Sustainability Initiative (GeSI) and Carbon Trust. *ICT Guidance on GHG Protocol Product Life Cycle Accounting and Reporting Standard*. https://www.gesi.org/public-resources/ict-guidance-on-ghg-protocol-product-life-cycle-accounting-and-reporting-standard/. Accessed 2 Jul 2025. Nov. 2024.

[119] Lars Andringa. "Estimating energy consumption of Cloud-Native applications". PhD thesis. 2024.

[120] Hubblo. *Scaphandre: Energy consumption monitoring agent*. https://github.com/hubblo-org/scaphandre. Accessed: 2025-06-24. 2025.

[121] Hubblo. *Overflow in energy counter can lead to wrong power measurements*. https://github.com/hubblo-org/scaphandre/issues/280. GitHub issue #280, *hubblo-org/scaphandre*. Feb. 2024. (Visited on 06/25/2025).

[122] k3s-io. *k3s-ansible: Ansible playbook for deploying K3s clusters*. https://github.com/k3s-io/k3s-ansible. Accessed: 2025-01-05. 2025. URL: https://github.com/k3s-io/k3s-ansible.

[123] Prometheus Community. *Prometheus Helm Charts*. Accessed: 2025-01-05. 2025. URL: https://github.com/prometheus-community/helm-charts.

[124] Colin Ian King. *stress-ng: A Linux System Stressing Tool*. https://github.com/ColinIanKing/stress-ng. Accessed: 2025-01-27. 2023.

[125] Jens Axboe. *fio: Flexible I/O Tester*. https://fio.readthedocs.io/en/latest/. Accessed: 2025-01-27. 2023.

[126] The ESnet Project. *iperf3: A TCP, UDP, and SCTP Network Bandwidth Measurement Tool*. https://iperf.fr/. Accessed: 2025-01-27. 2023.

[127] Caspar Wackerle. *Tycho: an accuracy-first container-level energy consumption exporter for Kubernetes (based on Kepler v0.9)*. GitHub repository. 2025. URL: https://github.com/casparwackerle/tycho-energy.