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

List of Figures

List of Tables

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

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
REVISE ENTIRE CHAPTER LATER XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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 **??**; 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 **??**, 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.

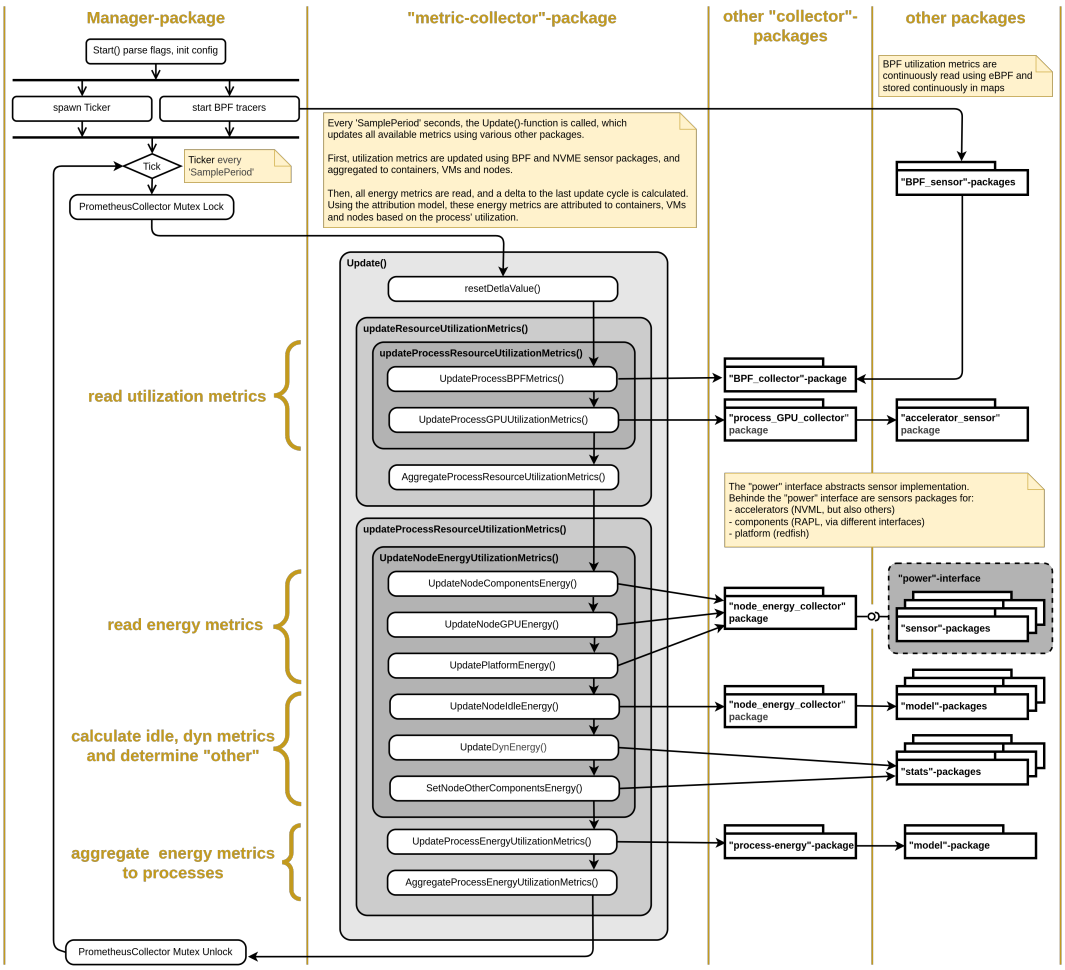


FIGURE 2.1: Kepler's synchronous update loop, where all collectors run at a unified sampling interval.

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.

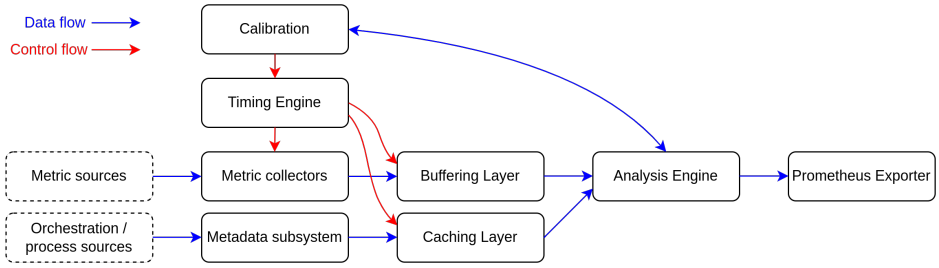


FIGURE 4.1: Subsystem Architecure, Dataflow and Control Flow

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.

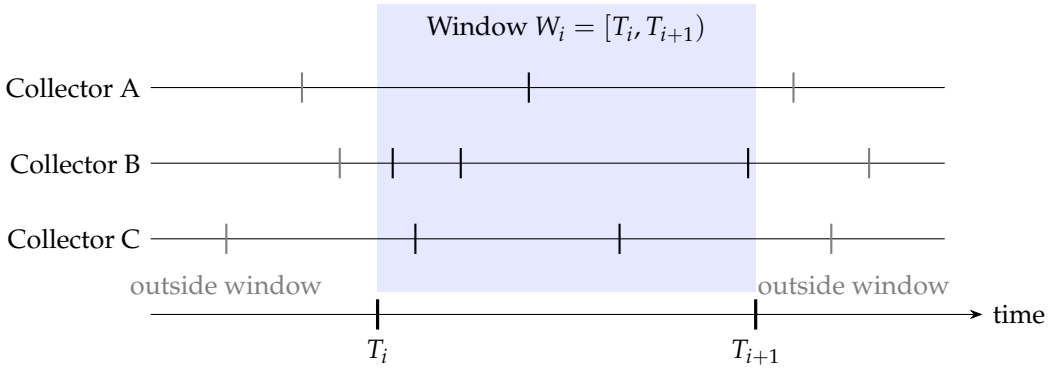


FIGURE 4.2: Analysis window W_i in relation to collectors

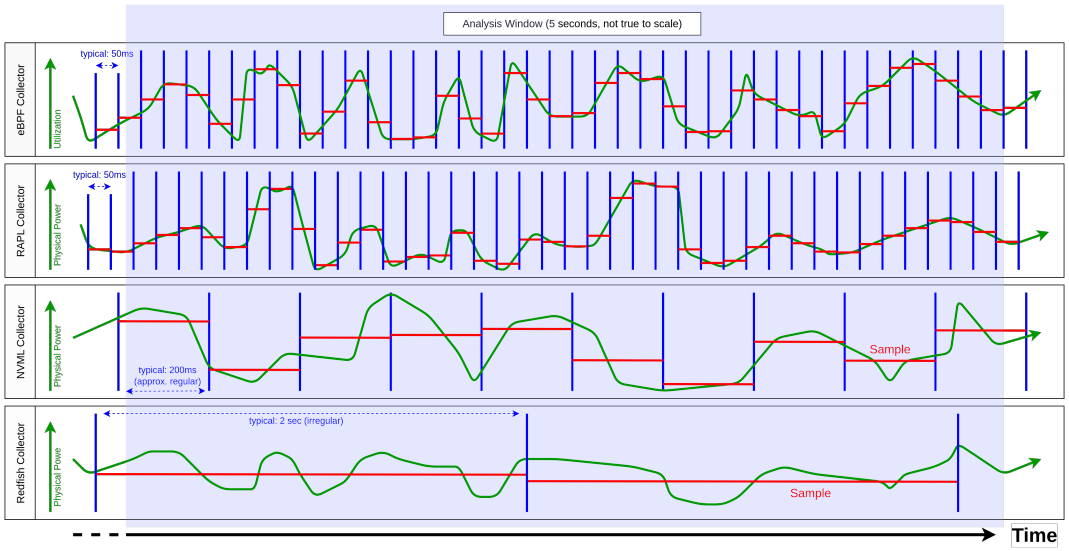
4.4.4 Comparison to Kepler Timing Model

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.

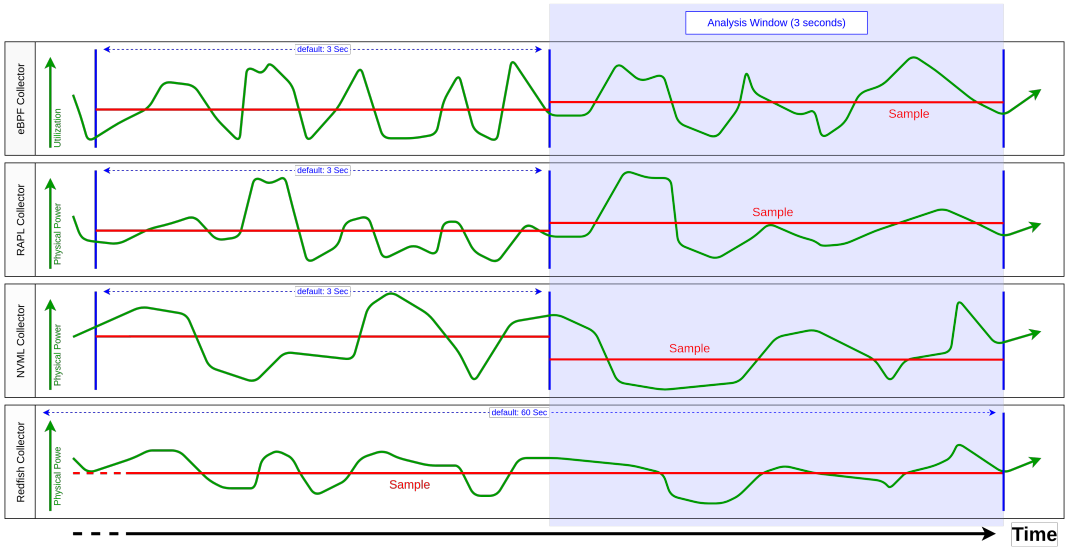
Tycho diverges fundamentally: collectors run independently, analysis windows are defined by attribution triggers rather than poll cycles, heterogeneous update patterns are supported natively, and export occurs immediately after each attribution step. This structure enables finer temporal resolution, 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.

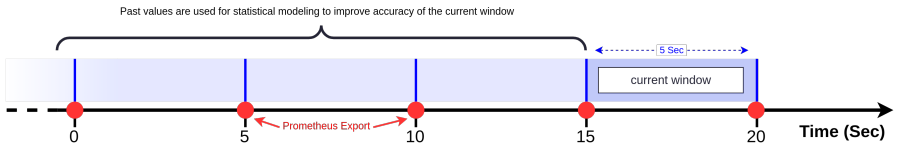


(A) Tycho Timing Model

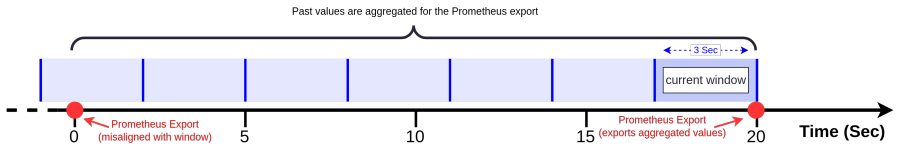


(B) Kepler Timing Model

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

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 and is decoupled from the event timing itself.

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

- *Event-driven metrics* capture transitions in processor ownership. They record the exact time at which execution begins or ends for a given context and therefore define precise temporal boundaries for attribution windows.
- *Cumulative counters* accumulate activity or duration over time and expose their values when queried. Their effective resolution is determined by the polling interval rather than by internal update frequency.
- *Quasi-instantaneous counters* sample hardware performance state at activity boundaries. Although observed at discrete points, their semantics remain tied to the execution periods they describe.

Two architectural properties follow directly from this structure. First, event-driven updates introduce no domain delay and therefore require no delay calibration. Second, the collector may operate at an arbitrary polling cadence, since temporal alignment is guaranteed by the event timestamps themselves. In Tycho the default interval is chosen to align with the RAPL window for convenience rather than correctness. Each event also carries the current container context, ensuring correct attribution when workloads migrate across control groups.

Within Tycho's temporal model, this domain occupies a special role. It accumulates fine-grained ownership information at execution boundaries and exposes it at analysis windows without influencing their timing. The resulting signals form a

complete temporal partition of CPU activity within each interval, supporting proportional attribution and reducing uncertainty in downstream energy modelling. Architecturally, the eBPF domain therefore provides the most precise and temporally coherent view of processor utilisation 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. These counters advance independently of Tycho's sampling schedule and describe the continuous energy behaviour of the processor.

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 entirely 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 synchronised with Tycho's timing engine so that each interval contains exactly one cumulative reading per domain. No delay calibration is required: 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 and stable decomposition of CPU energy that is preserved across intervals. This decomposition 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. Unlike in-band sources such as RAPL or eBPF telemetry, Redfish publishes instantaneous power values at coarse and irregular intervals determined entirely by the BMC implementation. These updates are asynchronous with respect to Tycho's timing engine and cannot be controlled or accelerated by the system.

Within Tycho's architecture, Redfish is therefore treated as a *latently published external observation* rather than a synchronisable metric source. Sampling is performed at fixed tick boundaries using the global monotonic timebase, but the temporal authority remains with the BMC. Repeated values are common, and new measurements may appear only after several ticks. Redfish thus does not define time; it constrains 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. It allows downstream analysis to

reason about the temporal reliability of each reading without assuming regular publication or low latency.

The irregular nature of Redfish publication also requires continuity guarantees. 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 carry the same timestamping and freshness semantics as true updates but do not indicate new power measurements.

Despite its limited temporal resolution, Redfish serves as Tycho's authoritative source for total node power. Its measurements provide a stable reference against which CPU- and accelerator-level energy estimates can be interpreted. Architecturally, Redfish complements fine-grained in-band domains by anchoring the system's global energy view, while its coarse and irregular behaviour is accommodated through explicit timestamping, freshness annotation, and controlled continuation rather than through high-frequency sampling or delay correction.

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 (§ **??**).

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.

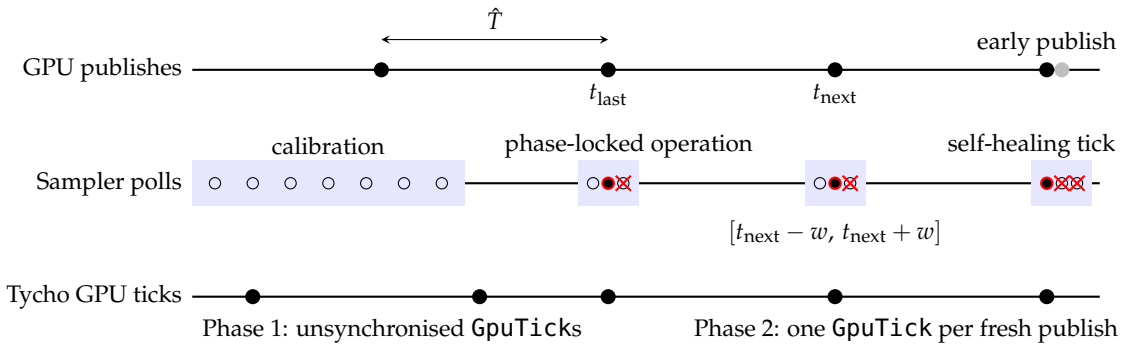


FIGURE 4.5: Phase-aware GPU polling timeline

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 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.

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 overlapping analysis windows and are removed automatically thereafter.

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 required for GPU and Redfish power metrics, whose firmware- or BMC-controlled publication intervals are stable in expectation but may exhibit non-negligible variability and are not formally 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.

No polling-frequency calibration is required for 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 to Tycho as static configuration parameters and are used by the analysis subsystem to align workload phases with metric data and to prevent premature 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.

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 requires interdependent derived quantities, imposing strict ordering constraints. Finally, analysis must remain online, deterministic, and reproducible, 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 § 4.8.6.

4.8.1.2 Conceptual Model

Analysis is structured 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.

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

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

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 observation. 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 makes 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 Metric Materialization and Intra-Cycle Visibility

4.8.2.1 Problem Statement

Later stages of the analysis pipeline operate on quantities derived from earlier analysis results. Without an explicit notion of when such quantities become available and how long they remain valid, dependency handling would be implicit and error-prone. Attribution and decomposition therefore require access to previously computed, window-scoped quantities without reinterpreting raw observations.

4.8.2.2 Materialization Model

Derived metrics are treated as *materialized facts* scoped to a single attribution window W_k. A metric is materialized exactly once per cycle as the result of a deterministic transformation applied within that window. Once materialized, the metric is immutable and represents the complete contribution of its defining semantics over W_k.

Materialized metrics form a directed dependency structure within the cycle. Some metrics consume only raw observations, while others consume previously materialized metrics. Dependencies may exist across stages or between metrics operating at the same semantic level but drawing on different inputs from the same window.

4.8.2.3 Formal Properties and Visibility

For a given cycle k, a metric is uniquely identified by its identity, label set, and the window W_k. At most one value exists for any such identity. If a metric's defining computation cannot be performed under the available evidence, the metric is undefined for that window and no value is materialized.

Materialization is strictly window-scoped. Metrics produced in cycle k are visible only within that cycle and do not persist as implicit inputs to future cycles. Any

dependence on prior windows must be mediated through explicit state, not through materialized metrics.

All materialized metrics are conceptually visible throughout the cycle. Correctness relies on strict execution order: metrics that depend on other metrics must be evaluated only after their dependencies have been materialized. Metrics that observe undefined inputs may themselves become undefined or produce degraded outputs according to their semantics, and such degradation propagates downstream.

4.8.2.4 Failure and Degradation Semantics

Metric computation is best-effort. If a metric cannot be computed for a given window, it is undefined for that window. Downstream metrics may abstain from producing results or yield explicitly degraded outputs, but the pipeline does not fabricate values to preserve completeness. This behavior prioritizes internal consistency, conservation, and temporal correctness over forced coverage.

4.8.2.5 Architectural Consequences

Metric materialization establishes a clear boundary between raw observation and derived analysis state. It enables later stages to operate on window-scoped quantities as first-class inputs, supports deterministic ordering across complex dependency structures, and prevents hidden coupling to collection or buffering. Once a cycle completes, its materialized metrics represent the maximal consistent interpretation achievable for that window and are not revised.

* - Asynchrony is expected and preserved.

4.9.2 Alternative Attribution Strategies

4.9.3 Complexity vs Accuracy Considerations

4.10 Summary

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–§ 5.9).

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.

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.

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.

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
`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.
`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

5.8 Correctness, Robustness, and Degradation Behavior

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.