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

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
REVISE ENTIRE CHAPTER LATER XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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.

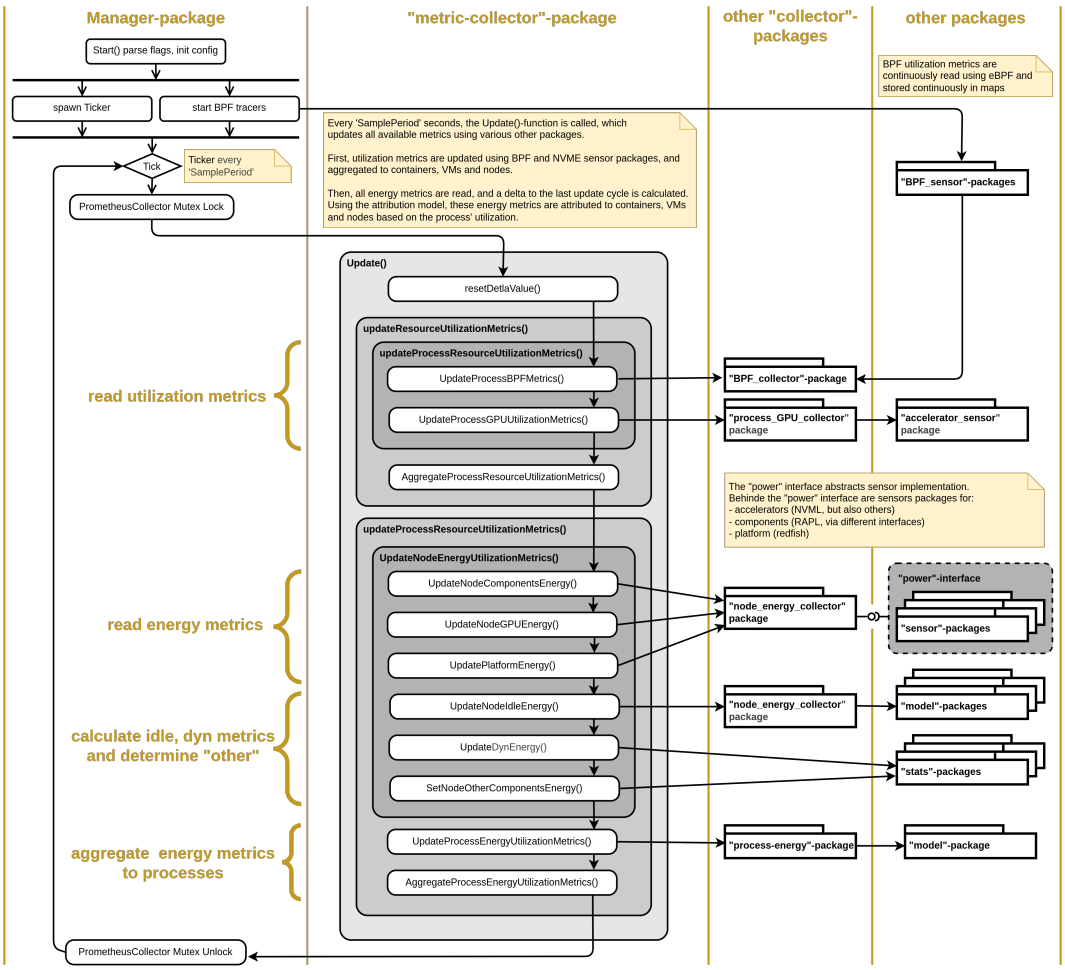


FIGURE 2.1: Kepler's synchronous update loop, where all collectors run at a unified sampling interval.

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.

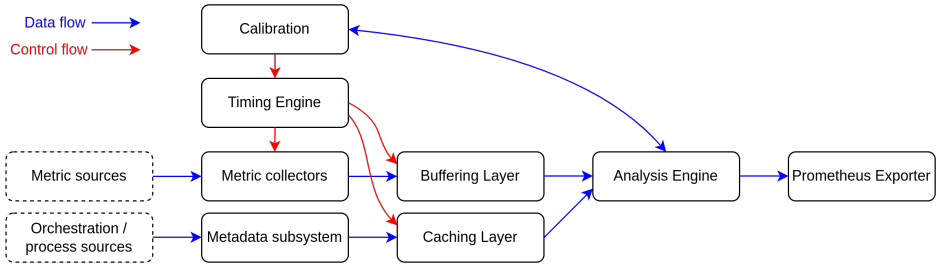


FIGURE 4.1: Subsystem Architecure, Dataflow and Control Flow

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.

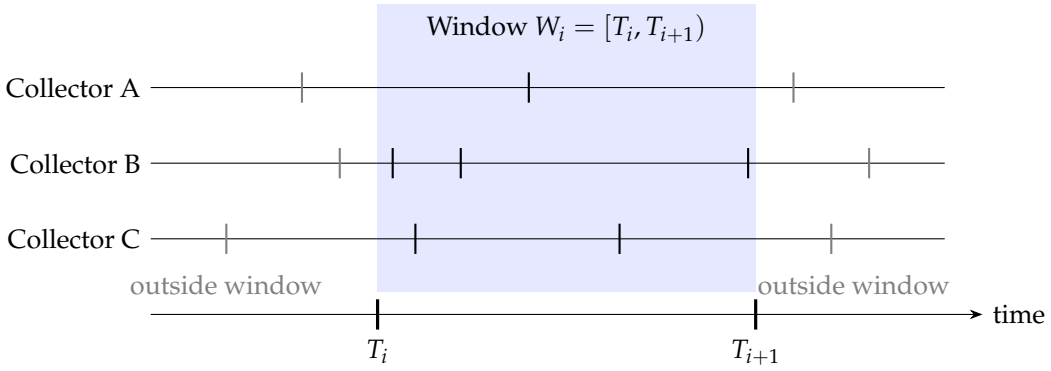


FIGURE 4.2: Analysis window W_i in relation to collectors

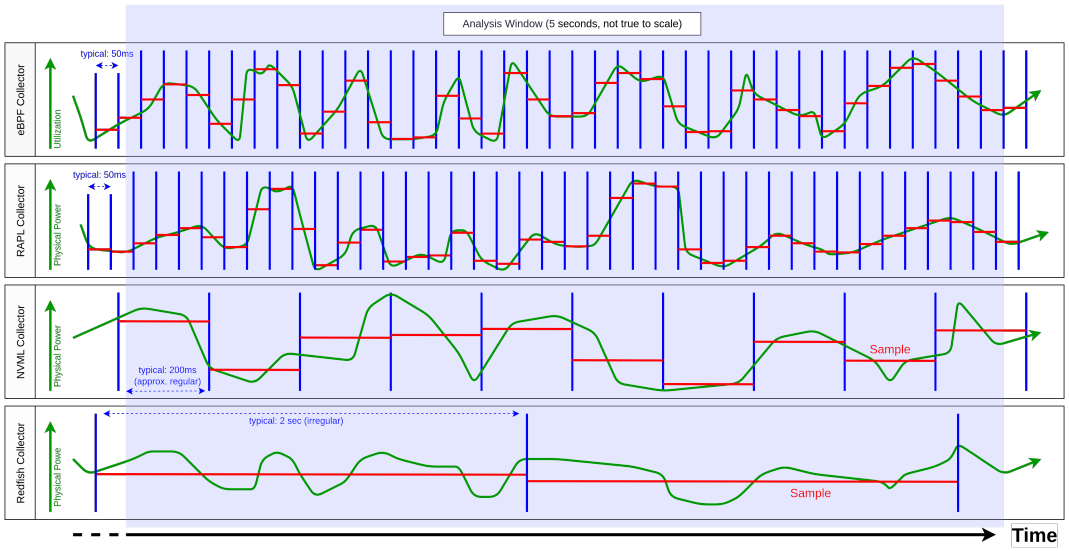
4.4.4 Comparison to Kepler Timing Model

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.

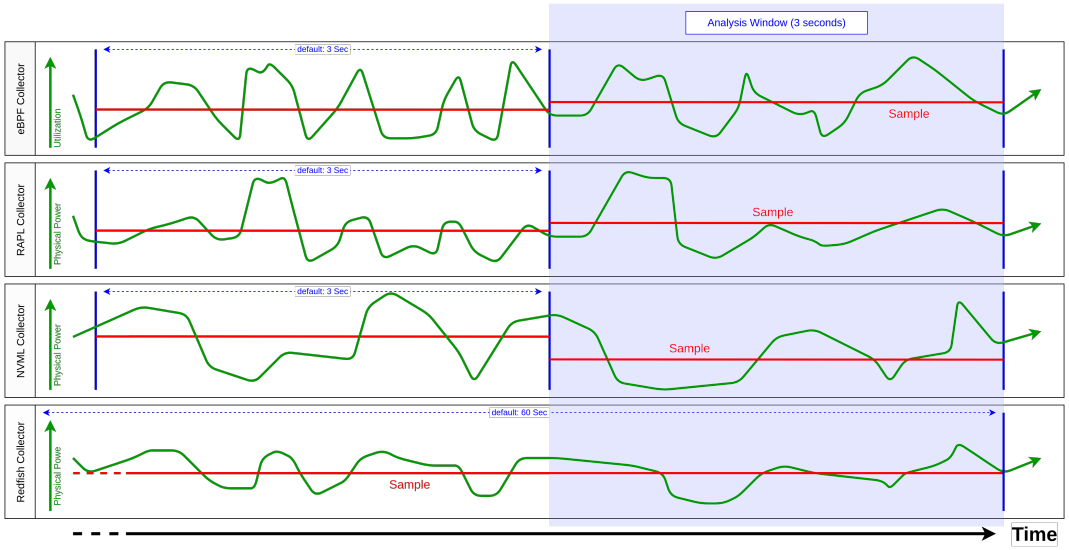
Tycho diverges fundamentally: collectors run independently, analysis windows are defined by attribution triggers rather than poll cycles, heterogeneous update patterns are supported natively, and export occurs immediately after each attribution step. This structure enables finer temporal resolution,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

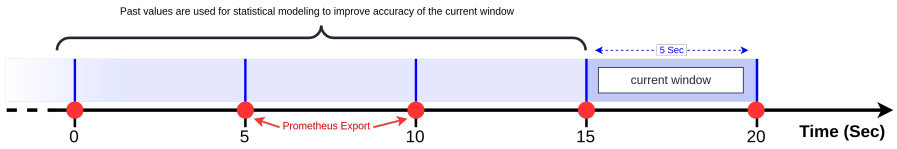


(A) Tycho Timing Model

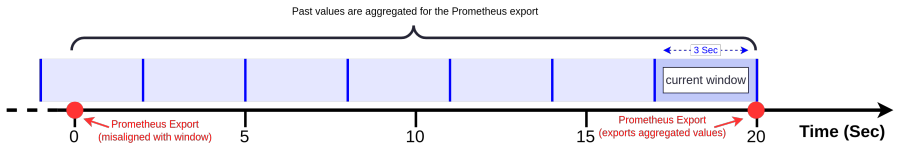


(B) Kepler Timing Model

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 (§ 6.7).

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.

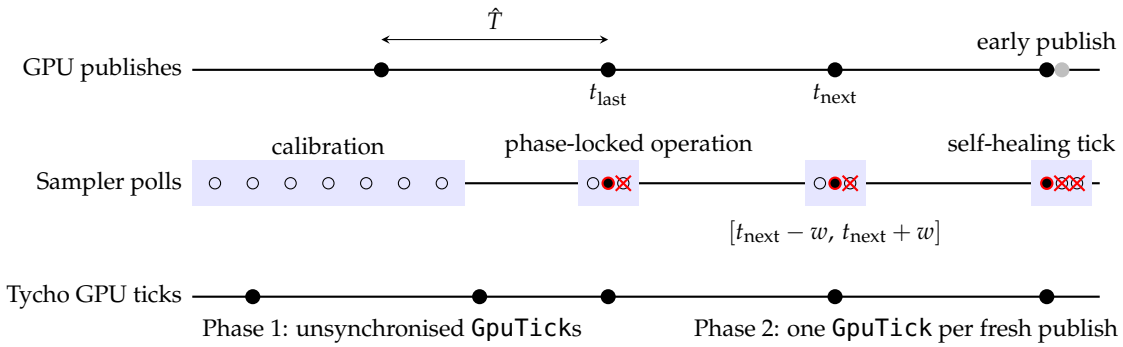


FIGURE 4.5: Phase-aware GPU polling timeline

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 (§ 6.8.1). 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 Purpose and Scope

4.8.2 Attribution Goals and Non-Goals

4.8.3 Analysis Engine as an Architectural Actor

4.8.4 Attribution Windows and Timebase Assumptions

4.8.5 Stage 1: Component Metric Construction

4.8.5.1 eBPF Utilization Metrics (Totals and Aggregates)

4.8.5.2 RAPL Domain Energy Metrics

4.8.5.3 Redfish-Corrected System Energy Metric

4.8.5.4 GPU-Corrected Energy Metric

4.8.6 Stage 2: System-Level Energy Model and Residual

4.8.6.1 Global Energy Decomposition

4.8.6.2 Residual Definition and Interpretation Limits

4.8.6.3 Negative Residuals as a Temporal Consequence

4.8.6.4 Conservation and Consistency Constraints

4.8.7 Stage 3: Idle and Dynamic Energy Semantics

4.8.7.1 RAPL Idle and Dynamic Semantics

4.8.7.2 Redfish-Corrected Idle and Dynamic Semantics

4.8.7.3 GPU Idle and Dynamic Semantics

4.8.8 Stage 4: Workload Attribution and Aggregation

4.8.8.1 Attribution Identity and Join Contract

4.8.8.2 CPU Dynamic Attribution to Workloads

4.8.8.3 CPU Idle Energy Allocation to Workloads

4.8.8.4 GPU Dynamic Attribution to Workloads

4.8.8.5 GPU Idle Energy Handling

4.8.8.6 Workload-Level Utilization and Energy Aggregates

4.8.9 Stability, Validity Conditions, and Architectural Limits

4.8.10 Architectural Consequences

4.9 Architectural Trade-Offs and Alternatives Considered

4.9.1 Alternative Timing Designs

* - 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.

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 (§ 5.7).

- **Export** observes the results of analysis and exposes them to external systems without influencing upstream execution or attribution semantics (§ 5.7).

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.

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.

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 6.1.

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. 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

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.

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 6.4 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 6.5.

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 § 6.3.2 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 caused by DVFS or thermal transitions 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.

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 6.2 and 6.3 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 Hierarchy Modeling

5.5.1.1 Node, Workload, Pod, Container Identities

5.5.1.2 Join Keys and Referential Stability

5.5.2 Identity Lifetime Management

5.5.2.1 Stability Within Attribution Windows

5.5.2.2 Controlled Evolution Across Windows

5.5.3 Degradation Under Metadata Incompleteness

5.5.3.1 Missing Joins and Fallback Semantics

5.6 Calibration Mechanisms

5.6.1 Polling-Frequency Calibration

5.6.1.1 Motivation and Correctness Role

5.6.1.2 Application to Collector Scheduling

5.6.2 Delay Calibration

5.6.2.1 Delay Estimation

5.6.2.2 Use of Calibrated Delays in Analysis

5.6.3 Calibration Failure Modes

5.6.3.1 Stale Calibration and Safety Constraints

5.7 Analysis and Attribution Pipeline

5.7.1 Pipeline Orchestration and Stage Execution

5.7.1.1 Stage Ordering and Dependencies

5.7.1.2 Per-Window Execution Contract

5.7.2 Stage 1: Component Metric Construction

5.7.2.1 Aligned Per-Window Inputs

5.7.2.2 eBPF Utilization Metrics (Totals and Aggregates)

5.7.2.3 RAPL Domain Energy Metrics

5.7.2.4 Redfish-Corrected System Energy Metric

5.7.2.5 GPU-Corrected Energy Metric

5.7.3 Stage 2: System-Level Energy Model and Residual

5.7.3.1 Global Energy Decomposition Realization

5.7.3.2 Residual Computation

5.7.3.3 Handling of Negative Residuals in Practice

5.7.3.4 Conservation and Consistency Checks

5.7.4 Stage 3: Idle and Dynamic Energy Semantics

5.7.4.1 RAPL Idle and Dynamic Realization

Chapter 6

Placeholder for WIP sections

6.1 System Environment for Development, Build and Debugging

This section documents the environment used to develop, build, and debug *Tycho*; detailed guides live in [56].

6.1.1 Host Environment and Assumptions

All development and debugging activities for *Tycho* were performed on bare-metal servers rather than virtualized instances. Development matched the evaluation target and preserved access to hardware telemetry such as RAPL, NVML, and BMC Redfish. The host environment consisted of Lenovo ThinkSystem SR530 servers (Xeon Bronze 3104, 64 GB DDR4, SSD+HDD, Redfish-capable BMC).

The systems ran Ubuntu 22.04 with a Linux 5.15 kernel. Full root access was available and required in order to access privileged interfaces such as eBPF. Kubernetes was installed directly on these servers using PowerStack[1], and served as the platform for deploying and testing *Tycho*. Access was via VPN and SSH within the university network.

6.1.2 Build Toolchain

Two complementary workflows are used: a dev path (local build, run directly on a node for interactive debugging) and a deploy path (build a container image, push to GHCR, deploy as a privileged DaemonSet via *PowerStack*).

6.1.2.1 Local builds

The implementation language is Go, using `go version go1.25.1` on `linux/amd64`. The `Makefile` orchestrates routine tasks. The target `make build` compiles the exporter into `_output/bin/<os>_<arch>/kepler`. Targets for cross builds are available for `linux/amd64` and `linux/arm64`. The build injects version information at link time through `LDFLAGS` including the source version, the revision, the branch, and the build platform. This supports traceability when binaries or images are compared during experiments.

6.1.2.2 Container images

Container builds use Docker Buildx with multi arch output for `linux/amd64` and `linux/arm64`. Images are pushed to the GitHub Container Registry under the project repository. For convenience there are targets that build a base image and optional variants that enable individual software components when required.

6.1.2.3 Continuous integration

GitHub Actions produces deterministic images with an immutable commit-encoded tag, a time stamped dev tag, and a latest for `main`. Builds are triggered on pushes to the main branches and on demand. Buildx cache shortens builds without affecting reproducibility.

6.1.2.4 Versioning and reproducibility

Development proceeds on feature branches with pull requests into `main`. Release images are produced automatically for commits on `main`. Development images are produced for commits on `dev` and for feature branches when needed. Dependency management uses Go modules with a populated `vendor/` directory. The files `go.mod` and `go.sum` pin the module versions, and `go mod vendor` materializes the dependency tree for offline builds.

6.1.3 Debugging Environment

The debugger used for *Tycho* is **Delve** in headless mode with a Debug Adapter Protocol listener. This provides a stable front end for interactive sessions while the debugged process runs on the target node. Delve was selected because it is purpose built for Go, supports remote attach, and integrates reliably with common editors without altering the build configuration beyond standard debug symbols.

6.1.3.1 Remote debugging setup

Debug sessions are executed on a Kubernetes worker node. The exporter binary is started under Delve in headless mode with a DAP listener on a dedicated TCP port. The workstation connects over an authenticated channel. In practice an SSH tunnel is used to forward the listener port from the node to the workstation. This keeps the debugger endpoint inaccessible from the wider network and avoids additional access controls on the cluster. To prevent metric interference the node used for debugging excludes the deployed DaemonSet, so only the debug instance is active on that host.

6.1.3.2 Integration with the editor

The editor is configured to attach through the Debug Adapter Protocol. In practice a minimal launch configuration points the adapter at the forwarded listener. Breakpoints, variable inspection, step control, and log capture work without special handling. No container specific extensions are required because the debugged process runs directly on the node.

The editor attaches over the SSH-forwarded DAP port; the inner loop is build locally with `make`, launch under Delve with a DAP listener, attach via SSH, inspect, adjust,

repeat. When the goal is to validate behavior in a cluster setting rather than to step through code, the deploy oriented path is used instead. In that case the image is built and pushed, and observation relies on logs and metrics rather than an attached debugger.

6.1.3.3 Limitations and challenges

Headless remote debugging introduces some constraints. Interactive sessions depend on network reachability and an SSH tunnel, which adds a small amount of latency. The debugged process must retain the privileges needed for eBPF and access to hardware counters, which narrows the choice of where to run sessions on multi tenant systems. Running a second exporter in parallel on the same node would distort measurements, which is why the DaemonSet is excluded on the debug host. Container based debugging is possible but less convenient given the need to coordinate with cluster security policies. For these reasons, most active debugging uses a locally built binary that runs directly on the node, while container based deployments are reserved for integration tests and evaluation runs.

6.1.4 Supporting Tools and Utilities

6.1.4.1 Configuration and local orchestration

A lightweight configuration file `config.yaml` consolidates development toggles that influence local runs and selective deployment. Repository scripts read this file and translate high level options into concrete command line flags and environment variables for the exporter and for auxiliary processes. This keeps day to day operations consistent without editing manifests or code, and aligns with the two workflows in § 6.1.2. Repository scripts map configuration keys to explicit flags for local runs, debug sessions, and ad hoc deploys.

6.1.4.2 Container, cluster, and monitoring utilities

Supporting tools: Docker, kubectl, Helm, k3s, Rancher, Ansible, Prometheus, Grafana. Each is used only where it reduces friction, for example Docker for image builds, kubectl for interaction, and Prometheus/Grafana for observability.

6.1.5 Relevance and Limitations

6.1.5.1 Scope and contribution

The development, build, and debugging environment described in § 6.1.2 and § 6.1.3 is enabling infrastructure rather than a scientific contribution. Its purpose is to make modifications to *Tycho* feasible and to support evaluation, not to advance methodology in software engineering or tooling.

Documenting the environment serves reproducibility and auditability. A reader can verify that results were obtained on bare-metal with access to the required telemetry, and can reconstruct the build pipeline from source to binary and container image. The references to the repository at the start of this section in § 6.1 provide the operational detail that is intentionally omitted from the main text.

6.1.5.2 Boundaries and omissions

Installation steps, editor-specific configuration, system administration, security hardening, and multi tenant policy are out of scope; concrete commands live in the repository. Where concrete commands matter for reproducibility they are available in the repository documentation cited in § 6.1.

6.2 eBPF Collector Integration

6.2.1 Purpose and Scope

The eBPF collector implements Tycho's kernel level acquisition of CPU activity data. It attaches to selected kernel events, accumulates per process and per CPU utilisation metrics, and exposes these values to userspace through shared maps. The userspace component periodically retrieves these aggregates, converts them into per tick deltas, and forwards them to the analysis pipeline. This subsystem provides fine grained measurements of task execution, interrupt handling, idle behaviour, and selected performance counters, forming the software side input required for CPU level energy attribution.

6.2.2 Kernel Instrumentation Overview

The eBPF subsystem consists of a set of kernel programs that record CPU activity in response to selected events. These programs update per CPU and per process data structures that hold runtime, interrupt durations, idle time, page cache counters, and basic performance information. All updates occur inside the kernel at the moment the corresponding events take place.

Scheduler related programs track on CPU durations of tasks and refresh their associated metadata, including container identifiers and task classification flags. Interrupt related programs account for time spent in hardware and deferred interrupts by recording entry and exit timestamps on each CPU. Additional programs record page cache access and writeback activity for the active task. Hardware performance counters are sampled through preconfigured performance monitoring units and appended to the task level aggregates.

The kernel side implementation is entirely event driven and maintains its own accounting structures without assistance from userspace. Userspace interacts with this subsystem only through periodic retrieval of the aggregated values.

6.2.3 Event Handling and Data Structures

The kernel programs update a small set of per CPU and per process data structures that hold all intermediate accounting information. Each CPU maintains a local state that stores the timestamp of the last activity change, the identity of the active task, and counters for idle and interrupt time. Per process aggregates reside in a bounded LRU map that holds accumulated runtime, performance counter deltas, page cache activity, and classification metadata.

Scheduler events drive the recording of task level runtime. When a task leaves the CPU, the program computes the elapsed duration since its previous activation and adds this value to the task entry in the process map. At the same point, hardware

performance counters for that task are sampled and the resulting deltas appended to its aggregates. The incoming task is then installed as the active task for that CPU and its start timestamp recorded.

Interrupt events update the per CPU bins for hardware and deferred interrupts. Entry handlers record a timestamp and exit handlers apply the difference to the local interrupt counters. During periods where no task is running, the CPU state marks the idle condition and accumulates the corresponding time until the next activity change.

Page cache programs increment per process counters whenever the active task performs relevant cache or writeback operations. These counters accumulate until the userspace collector retrieves and resets them.

All updates occur in per CPU or LRU maps to avoid cross core contention. The kernel side stores only aggregate values and does not perform any aggregation across CPUs or processes; consolidation is deferred to userspace.

6.2.4 Userspace Collector Logic

The userspace collector retrieves kernel aggregates at a fixed polling interval and converts them into per tick data structures for the analysis pipeline. At each interval it performs batched lookups on the per process map and per CPU bins, extracting and deleting all entries in a single operation. This yields the cumulative values recorded since the previous poll.

For each process entry, the collector computes deltas relative to the values reported in the previous tick and attaches the current container identifier and task classification flags exported by the kernel. Per CPU bins for idle time and interrupt durations are read and reset during the same operation. All values are appended to a single tick record that contains the full set of process level and CPU level aggregates observed during that interval.

The collector stores the resulting tick in Tycho's ring buffer for later analysis and export. Short lived tasks may appear only once if they terminate before the next polling interval, and processes that do not accumulate measurable activity between polls do not produce new entries. The collector performs no reconstruction or inference and relies entirely on the values supplied by the kernel programs.

6.2.5 Collected Metrics

The kernel programs expose all accumulated values through per process and per CPU maps. At each polling interval the userspace collector retrieves these aggregates and converts them into per tick deltas that enter the attribution pipeline. The collected metrics cover time based activity, selected performance counters, and task classification information. Table 6.1 lists the complete set of metrics produced by the eBPF subsystem.

All counters are returned as cumulative values since the previous retrieval and are reset or replaced by fresh entries during the same operation. The userspace collector attaches the resulting deltas to a single tick structure that enters Tycho's analysis and export pipeline.

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 6.1: Metrics collected by the kernel `eBPF` subsystem.

6.2.6 Performance, Overheads, and Stability

The kernel programs are designed to operate with minimal overhead on all supported workloads. All high frequency updates occur in per CPU maps, which avoids cross core contention and removes the need for locking inside the event handlers. Per process aggregates are stored in a bounded LRU map that limits memory usage and evicts inactive entries automatically. Each event handler performs only timestamp arithmetic and counter updates and does not allocate memory or invoke complex helper functions.

Userspace polling employs batched lookup and deletion to minimise system call overhead and maintain constant retrieval cost regardless of the number of active tasks. Performance monitoring units are preconfigured during collector initialisation and remain active for the lifetime of the program, which avoids repeated setup work during event handling. The collector processes all kernel aggregates in a single pass per interval and writes the resulting tick directly into the ring buffer without additional synchronisation.

The subsystem includes several safeguards to ensure stable operation across kernel versions. CO RE based field resolution protects access to task structures with

varying layouts, and all kernel programs use fixed size maps with explicit bounds to prevent overwrites. Cgroup identifiers and task classification flags are exported directly from scheduler events to ensure consistent attribution. The implementation handles idle threads and kernel threads explicitly and resets per CPU bins after each polling interval to avoid carryover between ticks.

6.2.7 Limitations

The eBPF subsystem exposes only the metrics supported by the attached kernel programs and hardware counters. It does not record processor frequency changes or C state transitions, and these values are therefore not available to downstream analysis. Hardware performance counters are sampled at task boundaries and may not reflect activity that occurs between consecutive events.

Short lived processes may terminate before the next polling interval and can therefore appear only once or not at all in the collected data. Tasks that accumulate no measurable activity between polls do not produce updates. Under workloads with very high interrupt activity, per CPU bins may grow rapidly, although they remain bounded by the polling interval.

All metrics depend on the availability of kernel events and may vary across kernel versions or configurations. If a kernel does not provide certain tracepoints or PMU events, the corresponding metrics remain unavailable in the exported tick data.

6.3 GPU Collector Integration

6.3.1 Introduction and Motivation

Accelerators are increasingly responsible for the energy footprint of modern compute workloads. To attribute this consumption to containerized applications with high temporal accuracy, Tycho must incorporate GPU telemetry into the same unified timing model used for RAPL, eBPF, and Redfish domains (§ 6.7). Achieving this integration is challenging: GPU drivers do not expose continuous measurements but publish telemetry at discrete, hardware-dependent intervals. If these intervals are not respected, sampling quickly suffers from aliasing, redundant reads, and temporal drift across subsystems, as well as imprecise timing.

NVIDIA's telemetry interfaces further complicate accurate measurement. The widely used `nvmlDeviceGetPowerUsage` function reports a *one-second trailing average*[57], not the instantaneous power required for sub-second energy attribution. High-frequency power samples are available only through specialised field APIs. Cumulative energy counters (when present) provide authoritative publish boundaries, but they are absent on many devices, including consumer GPUs and MIG configurations. Process-level telemetry is even more restrictive: NVML aggregates utilisation over caller-specified wall-clock windows and provides no information about the device's internal publish cadence.

Because of these structural limitations, fixed polling intervals or naïve periodic sampling are fundamentally insufficient. Accurate attribution requires that Tycho (i) infer the GPU's implicit publish cadence, (ii) align its sampling with this cadence,

and (iii) integrate both device- and process-level telemetry into the global measurement timeline without violating the strict monotonic ordering enforced by Tycho's multi-domain ring buffer (§ 6.8.1).

This work introduces two contributions that address these challenges:

- **A phase-aware sampling mechanism** that infers the GPU's hidden publish rhythm and adaptively concentrates polling around predicted update edges. This transforms GPU sampling from periodic polling into a timing-aligned, event-driven process.
- **A unified integration of GPU telemetry** into Tycho's global timebase, producing at most one `GpuTick` per confirmed hardware update, with timestamps that are directly comparable to all other energy domains.

Together, these mechanisms provide temporally precise, low-latency GPU measurements while respecting the variability and constraints of NVIDIA's telemetry ecosystem. This elevates the GPU subsystem to a first-class energy domain in Tycho and enables accurate container-level attribution in heterogeneous accelerator environments.

6.3.2 Architectural Overview

The GPU collector is organised as a layered subsystem that integrates vendor telemetry, adaptive timing, and unified buffering into a coherent measurement pipeline. Its structure reflects Tycho's core design principles: strict adherence to a monotonic timebase, decoupling of heterogeneous sampling frequencies, and event-driven integration into the platform-wide timing and buffering infrastructure (§ 6.7, § 6.8.1).

At the lowest layer, the collector interfaces with NVIDIA accelerators through a backend abstraction compatible with both `NVML` and `DCGM` ("*Data Center GPU Manager*"). This abstraction handles device enumeration, capability probing, MIG topology inspection, and access to device and process telemetry. The collector does not assume uniform backend capabilities: cumulative energy counters, instantaneous power fields, and process-level utilisation may or may not be available depending on hardware generation and configuration.

Above this backend, the collector exposes two measurement paths:

- **Device path.** Retrieves power, utilisation, frequency, thermal, and memory metrics for all devices and MIG instances. These values describe the instantaneous operational state of the accelerator.
- **Process path.** Aggregates per-process utilisation over a backend-defined wall-clock window. This enables multi-tenant attribution but is inherently retrospective and independent of the device's internal publish cadence.

Both paths feed into a shared sampling layer governed by Tycho's timing engine. The device path is triggered by a *phase-aware scheduler* that aligns its polling activity with the driver's implicit publish cadence. The process path is invoked in lockstep with the device polling loop. Process windows advance with each successful query, but only samples associated with confirmed device updates are propagated

into `GpuTick` events, which keeps the exported timeline aligned with the device publishes.

The final integration step mirrors all other Tycho subsystems: each confirmed hardware update is converted into a `GpuTick` structure containing device and (optionally) process snapshots, together with a strictly ordered monotonic timestamp. This tick is emitted into Tycho's multi-domain ring buffer, where it becomes part of the unified energy timeline used for correlation and attribution across eBPF, RAPL, and platform power domains.

Figure 4.5 (later in this section) provides a conceptual overview of this pipeline, illustrating the interaction between the backend interface, the phase-aware sampler, and Tycho's global collection engine.

6.3.3 Phase-Aware Sampling: Conceptual Overview

GPU drivers publish power and utilisation metrics at discrete, device-internal intervals. These updates occur neither continuously nor synchronously with the sampling frequencies required by Tycho's timing engine. Because the driver does not expose its publish cadence directly, a naïve fixed-interval polling strategy risks both *aliasing* (missing updates) and *redundancy* (repeatedly reading identical values). Either effect would distort the temporal alignment of GPU measurements with the rest of Tycho's energy domains.

To avoid this, the GPU collector introduces a *phase-aware sampling* mechanism that infers the driver's implicit publish cadence from observations. The sampler tracks two quantities: an estimated publish period and the phase offset between the device's update rhythm and Tycho's monotonic timebase. By predicting the next likely update moment, the sampler can modulate its polling intensity accordingly:

- **Base mode:** low-frequency polling maintains coarse alignment and detects long-term drift in the publish cadence.
- **Burst mode:** when the current time approaches a predicted update edge, the sampler briefly increases its polling frequency to minimise the latency between the hardware update and Tycho's observation of it.

This adaptive strategy ensures that Tycho reads the device only when a fresh publish is likely to be available. Freshness is determined by comparing each snapshot to the most recent confirmed update, preferably via cumulative energy counters when present, or otherwise via power deltas exceeding a configurable threshold. Only when a new publish is detected does the sampler emit an event.

The resulting behaviour is simple but powerful:

> *Each hardware update produces at most one `GpuTick`, and no tick is emitted unless the device has genuinely updated.*

This one-to-one correspondence is critical for integrating GPU measurements into Tycho's unified energy timeline. It guarantees temporal fidelity, eliminates redundant samples, and ensures that GPU metrics are directly comparable with other measurements obtained under the same timing and buffering semantics.

The next subsection formalises this behaviour by presenting the timing model used to estimate publish periods, track phase offsets, and define the burst window around predicted update edges.

6.3.4 Phase-Aware Timing Model

The timing model enables Tycho to infer the GPU driver's implicit publish cadence and to align sampling with the actual update moments of the hardware. It maintains two quantities derived from confirmed device updates: an estimate of the *publish period* and a *phase offset* relative to Tycho's monotonic clock. This subsection presents the model in a unified mathematical form.

Let $t_{\text{obs},k}$ denote the monotonic timestamp of the k-th confirmed hardware update. From these observations, the sampler derives the period estimate $\hat{T}_k$ and phase estimate $\hat{\phi}_k$.

Period Estimation. Each new inter-update interval

$$\Delta t_k = t_{\text{obs},k} - t_{\text{obs},k-1}$$

provides a direct sample of the device's publish period. To remain robust to jitter caused by DVFS, thermal transitions, or backend noise, the sampler applies an exponential moving average (EMA):

$$\hat{T}_k = (1 - \alpha_T)\, \hat{T}_{k-1} + \alpha_T\, \Delta t_k,$$

with $\alpha_T \in (0, 1)$ controlling the smoothing strength. The resulting estimate is clamped to a stable range derived from Tycho's engine cadence, ensuring predictable behaviour across different GPUs.

Phase Tracking. Given a current period estimate, the expected time of the k-th update is

$$\hat{t}_k = t_{\text{obs},k-1} + \hat{\phi}_{k-1} + \hat{T}_k.$$

The deviation

$$\delta_k = t_{\text{obs},k} - \hat{t}_k$$

represents the phase error. The sampler updates its phase estimate through a second EMA:

$$\hat{\phi}_k = (\hat{\phi}_{k-1} + \alpha_\phi\, \delta_k) \bmod \hat{T}_k,$$

where α_ϕ is a small adaptation constant. This ensures smooth convergence toward the device's true publish rhythm.

Edge Prediction. At an arbitrary time t_{now}, the predicted next update edge is

$$t_{\text{next}} = t_{\text{obs},k} + n \cdot \hat{T}_{k+1} + \hat{\phi}_{k+1},$$

where n is the smallest non-negative integer such that $t_{\text{next}} \geq t_{\text{now}}$. This prediction determines where sampling effort should be concentrated.

Burst Window. To avoid continuous high-frequency polling, the sampler restricts hyperpolling to a narrow window of half-width w around t_{next}:

$$\text{mode}(t_{\text{now}}) = \begin{cases} \text{burst}, & |t_{\text{now}} - t_{\text{next}}| \leq w, \\ \text{base}, & \text{otherwise.} \end{cases}$$

The width w is expressed as a fraction of the calibrated engine cadence, ensuring proportional behaviour across platforms.

Summary. The phase-aware model enables Tycho to infer the GPU's implicit publish cadence solely from observed updates and to align sampling with the device's true update edges. By combining smooth period estimation, adaptive phase correction, and narrow burst windows around predicted publishes, the sampler detects new hardware updates with low latency and emits st most one `GpuTick` per publish.

Figure 4.5 visualises the behaviour of the phase-aware sampling model introduced above. The top lane represents the hardware's implicit publish sequence; the middle lane shows Tycho's adaptive polling pattern during both calibration and the phase-locked regime; and the bottom lane shows the resulting GPU ticks, demonstrating the one-to-one mapping between fresh device updates and emitted `GpuTick` events.

6.3.5 Event Lifecycle

The GPU collector converts each confirmed hardware update into a monotonically-timestamped `GpuTick` that integrates into Tycho's multi-domain energy timeline. The lifecycle consists of five stages: polling, device acquisition, optional process acquisition, freshness detection, and tick emission.

1. Poll Initiation. Polling is triggered solely by the phase-aware scheduler (§ 6.3.3, § 6.3.4). Base-mode polls track long-term cadence drift; burst-mode polls densely probe the vicinity of predicted update edges. Each poll receives a monotonic timestamp t_{now} that anchors the resulting event.

2. Device Snapshot Acquisition. A poll retrieves device-level telemetry for all GPUs and MIG instances, capturing power, utilisation, clocks, thermals, and memory state. All values reflect the device's instantaneous condition at t_{now} and form a consistent cross-device snapshot of the accelerator subsystem.

3. Optional Process Snapshot Acquisition. If available, process-level telemetry is sampled over a backend-defined wall-clock window (§ 6.3.6). Although retrospective, these samples are associated with the same monotonic timestamp as the device snapshot, ensuring that device and process data remain correlated without temporal ambiguity.

4. Freshness Determination. The collector compares the new device snapshot with the most recent confirmed update. Cumulative energy counters, when available, serve as the authoritative freshness signal; otherwise Tycho uses a power-delta threshold to avoid counting noise as updates. Only fresh snapshots update the period and phase estimators and proceed to the next stage.

5. Tick Emission. A fresh observation is converted into a `GpuTick` containing device and (optional) process snapshots and the timestamp t_{now}. The tick is then delivered to Tycho's multi-domain ring buffer (§ 6.8.1). If no fresh update is detected, the poll produces no tick, ensuring that the GPU timeline faithfully reflects the hardware's publish cadence.

Summary. The event lifecycle ensures that GPU telemetry is sampled only when meaningful, timestamped consistently with Tycho's timebase, and integrated without blocking or duplication. Each hardware update generates at most one `GpuTick`, providing a precise, causally ordered input to Tycho's cross-domain energy attribution pipeline.

6.3.6 Per-Process Telemetry Window

Device-level metrics describe the instantaneous state of each GPU, but many applications require attributing accelerator activity to individual processes or containers. NVIDIA's interfaces provide such information only in the form of *aggregated utilisation over a caller-specified time window*. Correctly selecting and interpreting this window is essential for obtaining meaningful per-process data and for aligning process-level records with the device-level timeline maintained by Tycho.

Wall-Clock Semantics. Unlike device publishes, which occur on the GPU's internal cadence, NVIDIA's per-process APIs integrate utilisation over a duration supplied by the caller. These interfaces expect a *wall-clock* interval, expressed in milliseconds, rather than a duration derived from Tycho's monotonic timebase. The distinction is crucial: Tycho's monotonic clock operates on an internal quantum chosen to support high-resolution scheduling (§ 6.7), but this quantum has no defined relationship to real elapsed time. Using monotonic differences directly would produce windows that are several orders of magnitude too short, yielding incomplete utilisation samples.

For this reason, Tycho maintains a separate wall-clock origin for each GPU or MIG instance. Whenever process telemetry is requested, the duration since the last successful query is computed using wall-clock time, ensuring that the backend receives a true real-time interval.

Window Derivation. For each owner (physical GPU or MIG instance), Tycho records the timestamp $t^{(i)}_{\text{last}}$ of the most recent successful process query. When a new query occurs at time $t^{(i)}_{\text{now}}$, the raw duration

$$\Delta t^{(i)}_{\text{raw}} = t^{(i)}_{\text{now}} - t^{(i)}_{\text{last}}$$

is transformed according to backend expectations:

1. *Clamping.* The duration is restricted to a safe range $\Delta t_{\min} \leq \Delta t^{(i)} \leq \Delta t_{\max}$ to avoid zero-length or excessively long sampling windows.

2. *Millisecond granularity.* NVIDIA's process APIs accept durations in whole milliseconds. Tycho therefore rounds the clamped value up to the next full millisecond to prevent systematic underestimation of utilisation.

After a successful query, the wall-clock origin is updated to $t_{\text{last}}^{(i)} \leftarrow t_{\text{now}}^{(i)}$, establishing continuity across successive sampling windows.

Temporal Alignment with Device Updates. Even though per-process telemetry describes accumulated activity rather than a snapshot, Tycho ensures that all process samples remain aligned with the device timeline. Each process record is associated with the device-level timestamp of the poll that triggered the query. If a device has never produced a fresh update, the collector uses the timestamp of the most recent device tick as the initial origin for its process window. This guarantees that device and process metrics are linked to the same global temporal reference and can be fused without interpolation.

Backend Variability and Robustness. Process-level support varies widely across NVIDIA hardware and software stacks. DCGM-capable systems typically expose high-quality, high-resolution utilisation data, whereas NVML-only systems (particularly consumer GPUs) may provide limited or noisy information. Tycho's design accommodates these differences gracefully: when a process query fails, the wall-clock origin is still advanced to prevent tight retry loops, and device-level sampling proceeds unaffected. This ensures stable behaviour even in mixed configurations where only a subset of devices expose meaningful per-process telemetry.

Summary. By separating wall-clock process windows from monotonic device timestamps and carefully aligning both within Tycho's timing architecture, the GPU collector provides process-level telemetry that is semantically correct, temporally consistent, and robust to backend limitations. This separation of concerns is essential for accurate multi-tenant attribution in heterogeneous accelerator environments.

6.3.7 Collected Metrics

The GPU collector reports two complementary categories of telemetry that together describe both the instantaneous state of each accelerator and the distribution of GPU activity across processes. All metrics are incorporated into a unified `GpuTick` structure and timestamped under Tycho's monotonic timebase, ensuring direct comparability with other domains.

Device-Level Metrics. Device and MIG-level metrics capture the operational state of the accelerator at the moment Tycho detects a fresh hardware update. These values include power, utilisation, memory usage, thermal data, and clock frequencies, along with backend-specific fields such as instantaneous power samples or cumulative energy counters. Cumulative energy, when available, is used as the authoritative indicator of publish boundaries and therefore plays a central role in the timing model and freshness detection. Due to a tendency of NVIDIA GPUs to still produce (invalid) values when queried for cumulative energy, the actual availability of correct cumulative energy-metrics is verified during the initial calibration. The collector uses a per-device validity mask derived from this calibration when performing freshness detection: cumulative energy is treated as authoritative only for devices whose counters are verified as monotonic, and other devices fall back to a power-delta threshold.

Process-Level Metrics. Process-level metrics describe the aggregated utilisation of individual processes over the backend-defined wall-clock window (§ 6.3.6). They enable multi-tenant attribution by associating GPU activity with specific applications, containers, or pods. Because these values represent accumulated work rather than an instantaneous snapshot, they are paired with the device-level timestamp of the triggering poll, ensuring temporal consistency within Tycho's unified timeline.

Tables 6.2 and 6.3 summarise the metrics collected at both levels.

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 6.2: 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 6.3: Process-level metrics collected over a backend-defined time window.

6.3.8 Configuration Parameters

The GPU collector exposes only a minimal set of configuration parameters. In contrast to traditional monitoring systems that require hand-tuned polling intervals, Tycho derives the parameters of the phase-aware sampler directly from the engine cadence calibrated during system startup (§ 6.7). This ensures that GPU sampling inherits the same temporal consistency as all other energy domains and remains robust across heterogeneous hardware.

The configuration governs three tightly coupled aspects of the sampling mechanism:

- **Cadence bounds.** The initial estimate of the GPU publish period, as well as its minimum and maximum permissible values, are expressed as simple fractions of the engine cadence. This constrains the period estimator to a stable range without relying on device-specific heuristics.
- **Polling intervals.** Both base-mode and burst-mode polling frequencies are derived from fixed ratios of the engine cadence. As a result, Tycho polls aggressively only when a publish is predicted without requiring manual tuning.
- **Burst-window width.** The half-width of the burst window around t_{next} is likewise tied to the engine cadence. This determines how narrowly the sampler focuses its hyperpolling effort around predicted publish edges.

Because all parameters scale with the calibrated cadence, the sampler adapts automatically to different GPU generations, backend behaviours, and platform timing characteristics. No user-facing configuration is required; temporal correctness follows directly from Tycho's system-wide timing model.

6.3.9 Robustness and Limitations

The GPU collector is designed to operate reliably across heterogeneous hardware, backend capabilities, and driver behaviours. Its phase-aware sampling, decoupled event queue, and unified timebase ensure that GPU telemetry integrates cleanly with

Tycho's multi-domain measurement framework. Nevertheless, several structural constraints in NVIDIA's telemetry ecosystem define the practical limits of what can be inferred and with what temporal precision.

Backend Variability. The capabilities of `NVML` and `DCGM` differ significantly across GPU generations and product classes. Datacenter GPUs typically expose cumulative energy counters, high-frequency instant power fields, and stable process-level utilisation, while consumer GPUs often lack cumulative energy and provide only coarse utilisation metrics. Tycho handles these differences gracefully (sampling continues even when certain fields are missing) but the quality of the resulting attribution reflects the capabilities of the underlying hardware.

Power Measurement Limitations. The widely used `nvmlDeviceGetPowerUsage` call provides a *one-second trailing average*, which is unsuitable as a high-frequency power signal. Tycho therefore relies on instantaneous power fields (e.g. field 186) when available, and uses cumulative energy counters as the authoritative freshness indicator. On devices lacking both instantaneous fields and cumulative energy, power-based freshness detection becomes less precise, increasing uncertainty in the inferred publish cadence.

Process Attribution Constraints. Process-level utilisation is inherently aggregated over a wall-clock window, since NVIDIA provides no access to per-process instantaneous state. This retrospective design imposes two limitations: (i) spikes shorter than the sampling window may be attenuated, and (ii) per-process values cannot be aligned to the exact moment of a device publish. Tycho addresses this by using the device-level timestamp to anchor all process records, but the granularity of attribution ultimately depends on backend resolution.

Cadence Inference and Jitter. Because the driver does not expose its publish cadence, Tycho must infer it indirectly. Under conditions of high load, thermal transitions, or DVFS-induced jitter, publish intervals may vary, introducing uncertainty into edge prediction. Tycho's EMA-based estimators maintain stability under such variability, but prediction accuracy is inherently bounded by the noisiness of the underlying telemetry.

Mixed and MIG Configurations. Systems combining MIG and non-MIG devices, or devices with partial telemetry support, may expose inconsistent field availability across accelerators. Cumulative energy counters may exist for some instances but not others; process information may be available only at the parent-device level. Tycho handles these cases through per-device fallbacks and independent cadence models, but the precision of multi-GPU attribution varies with the fidelity of each device's telemetry.

Vendor support scope. The current GPU collector supports only NVIDIA-based accelerators, following the design of Kepler. While this excludes other vendors, it is justifiable: according to market research[58], NVIDIA captured approximately 93% of the server GPU revenue in 2024. Given this dominant share, focusing on NVIDIA hardware is acceptable for the majority of data-centre GPU deployments.

6.4 RAPL Collector Integration

6.4.1 Purpose and Scope

The RAPL collector provides cumulative energy readings for all CPU related domains supported by the hardware. It retrieves these values once per tick and stores them in Tycho's ring buffer without applying intermediate processing. All interpretation, differencing, and model level handling occur in the downstream analysis layer.

6.4.2 Collector Mechanics and Data Flow

At each tick the collector is invoked by the timing engine to obtain a complete snapshot of the RAPL domains for all sockets. The collector queries the shared RAPL component, which exposes the cumulative hardware counters for package, core, uncore, and memory related domains when available. The results are placed into a `RaplTick` structure containing a monotonic timestamp and a per socket map of domain counters.

Only raw cumulative values are stored. The collector does not compute differences, apply scaling, or perform validity checks beyond verifying that the system exposes RAPL counters. Domain availability is determined by the underlying hardware, and unsupported domains are omitted from the per socket map. Each tick therefore contains exactly one set of cumulative counters per socket for all domains supported by that platform.

The resulting `RaplTick` is immutable and is written directly into Tycho's ring buffer, where it becomes available to the attribution and export components.

6.4.3 Exported Metrics

The metrics exported by the RAPL collector are shown in Table 6.4. All values represent cumulative energy since a hardware defined starting point and are reported once per tick.

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 6.4: Metrics exported by the RAPL collector per `RaplTick`.

6.4.4 Stability and Platform Variability

The collector records only the domains exposed by the hardware. Some platforms provide package level counters only, whereas others additionally expose core, uncore, or memory related domains. The collector does not approximate or reconstruct unsupported domains and stores only the counters returned by the system.

The cumulative counters obtained from RAPL are stable at the tick interval used by Tycho and do not exhibit missing data or inconsistent increments at this scale. Since the collector performs no differencing, scaling, or interpolation, it remains minimal and does not introduce additional sources of variability.

6.4.5 Limitations

Limitations of the RAPL collector arise solely from platform specific domain availability. Platforms that do not provide a particular domain omit it from the exported tick. Beyond these hardware differences the collector has no additional constraints, and all further processing occurs in the downstream analysis layer.

6.5 Redfish Collector Integration

6.5.1 Purpose and Scope

The Redfish collector retrieves chassis power readings from the Baseboard Management Controller at the global engine cadence. It performs exactly one Redfish query per tick, evaluates whether the returned value represents a new measurement, and emits either a fresh sample or a continuation sample. All results are timestamped with Tycho's monotonic clock and written into the shared ring buffer without further processing.

6.5.2 Collector Mechanics and Data Flow

At each tick the collector issues one HTTP request to the BMC and extracts the current power value together with metadata such as the chassis identifier, sequence number, and BMC timestamp when present. If the request fails, the collector skips the tick without producing a sample.

The collector maintains the last observed sequence number for each chassis. If the newly retrieved sequence number differs from the previous one, the measurement is classified as fresh and is emitted immediately. If no new sequence has appeared, the collector applies a heartbeat policy to ensure continuity of the power time series. The heartbeat policy evaluates whether the time since the last fresh sample exceeds a configurable threshold. If so, the collector emits a continuation sample that repeats the most recent power value and updates its metadata accordingly.

Tycho supports two heartbeat modes. In fixed mode the heartbeat threshold is defined explicitly via configuration parameters. In auto mode the collector tracks the inter-arrival times of fresh Redfish updates and derives the heartbeat threshold from the median of a sliding window over these intervals, bounded by conservative limits. Polling frequency is unaffected by either mode and always follows the engine tick.

Each emitted sample is placed into a Redfish record containing the chassis identifier, power value, sequence number, BMC timestamp when provided, monotonic collection time, and freshness information. The freshness field records the difference between the BMC time and the monotonic collection time. Continuation samples recompute freshness using the last available BMC timestamp.

The resulting Redfish record is immutable and is written directly into Tycho's ring buffer.

6.5.3 Collected Metrics

The collector emits one record per chassis whenever a fresh measurement is detected or a heartbeat continuation is required. The fields exported in each record are listed in Table 6.5.

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 6.5: Metrics collected by the Redfish collector.

Derived quantities such as energy are computed by downstream components and are not produced by the collector.

6.5.4 Stability and Behaviour Across Chassis

The collector handles multiple chassis independently. Each chassis maintains its own sequence history, heartbeat state, and sliding window of update intervals when auto mode is active. The collector does not attempt to synchronise readings across chassis and emits records for each chassis as they become available.

Freshness values may vary significantly across platforms due to differences in BMC timestamp accuracy and publication delays. The collector records these values without modification. Continuation samples recompute freshness using the last reported BMC timestamp, allowing downstream components to identify stale or delayed data.

6.5.5 Limitations

The precision and update rate of Redfish measurements are determined entirely by the underlying BMC implementation. Most BMCs publish power data at low and irregular frequencies, and some may provide timestamps with limited resolution or drift. The collector does not correct or filter these behaviours and relies on downstream analysis to account for timestamp quality or coarse temporal resolution. Redfish provides total chassis power only and does not expose component-level breakdowns.

6.6 Configuration Management

6.6.1 Overview and Role in the Architecture

Tycho adopts a simple, centralized configuration layer that is initialized during exporter startup and made globally accessible through typed structures. This layer defines all runtime parameters controlling timing, collection, and analysis behaviour. It serves as the interface between user-defined settings and the internal scheduling and buffering logic described in § 6.7.

The configuration is loaded once at startup, combining defaults, environment variables, and optional overrides passed through Helm or local flags. Its purpose is not to support dynamic reconfiguration, but to provide deterministic, reproducible operation across (experimental) runs. No backward compatibility with previous Kepler versions is maintained.

6.6.2 Configuration Sources

Configuration values can be provided in three ways: first, through a `values.yml` file during Helm installation, second, as command-line flags for local or debugging builds, and third, via predefined environment variables that act as defaults.

During startup, Tycho sequentially evaluates these sources in fixed order— defaults are loaded first, then environment variables, followed by any user-supplied overrides. The resulting configuration is stored in memory and printed once for verification. After initialization, all components reference the same in-memory configuration, ensuring consistent behaviour across collectors and analysis modules.

6.6.3 Implementation and Environment Variables

The configuration implementation in Tycho closely follows the approach used in Kepler v0.9.0. Each configuration key is mapped to an environment variable, which is resolved at startup through dedicated lookup functions. If no variable is set, the corresponding default value is applied. This mechanism enables flexible configuration without external dependencies or complex parsing logic. All variables are read once during initialization, after which they are cached in typed configuration structures. This guarantees consistent operation even if environment variables change later, since Tycho is not designed for live reconfiguration. The configuration layer is invoked before the collectors and timing engine are instantiated, ensuring that parameters such as polling intervals, buffer sizes, or analysis triggers are available to all components from the first cycle onward.

6.6.3.1 Validation and Normalization at Startup

During initialization, Tycho validates all user inputs and normalizes them to a consistent, safe configuration. First, basic bounds are enforced: the global timebase quantum must be positive, non-negative values are required for all periods and delays, and missing essentials fall back to minimal defaults. Trigger coherence is then checked. If `redfish` is selected while the Redfish collector is disabled, Tycho switches to the timer trigger and ensures a valid interval. Unknown triggers default to `timer`.

All periods and delays are aligned to the global quantum so that scheduling, buffering, and analysis operate on a common time grid. The analysis wait `DelayAfterMs` is raised if needed to cover the longest enabled per-source delay. Buffer sizing is derived from the slowest effective acquisition path (poll period plus delay) and the analysis wait, with a small safety margin. If Redfish is enabled, its heartbeat requirement is included to guarantee coverage. Sanity checks also ensure plausible Redfish cadence and warn if no collectors are enabled. Non-fatal environment hints (for example the RAPL powercap path) are reported at low verbosity.

The result is a single, internally consistent configuration snapshot. Adjustments are announced once at startup to aid reproducibility while avoiding log noise.

6.6.4 Evolution in Newer Kepler Versions

Subsequent Kepler releases (v0.10.0 and later) have replaced the environment-variable system with a unified configuration interface based on CLI flags and YAML files. This modernized approach simplifies configuration management and aligns better with Kubernetes conventions, providing clearer defaults and validation at startup.

Tycho intentionally retains the v0.9.0 model to maintain structural continuity with its experimental foundation. Since configuration handling is not a research focus,

adopting the newer scheme would add complexity without scientific benefit. Nevertheless, the newer Kepler design confirms that Tycho's configuration logic can be migrated with minimal effort if long-term maintainability becomes a requirement.

6.6.5 Available Parameters

All parameters are read at startup and remain constant throughout execution. The following table 6.6 summarizes the user-facing configuration variables with their default values and functional scope. Internal or experimental parameters are omitted for clarity.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
CHECK THIS TABLE BEFORE HANDIN, THIS NEEDS CLEANUP
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Variable	Default	Description
Collector enable flags		
`TYCHO_COLLECTOR_ENABLE_BPF`	`true`	Enables eBPF-based process metric collection.
`TYCHO_COLLECTOR_ENABLE_RAPL`	`true`	Enables RAPL energy counter collection.
`TYCHO_COLLECTOR_ENABLE_GPU`	`true`	Enables GPU power telemetry collection.
`TYCHO_COLLECTOR_ENABLE_REDFISH`	`true`	Enables Redfish-based BMC power collection.
Timing and delays		
`TYCHO_TIMEBASE_QUANTUM_MS`	`1`	Base system quantum (ms) defining the global monotonic time grid.
`TYCHO_RAPL_POLL_MS`	`50`	RAPL polling interval (ms).
`TYCHO_GPU_POLL_MS`	`200`	GPU telemetry polling interval (ms).
`TYCHO_REDFISH_POLL_MS`	`1000`	Redfish polling interval (ms); should be below BMC publish cadence.
`TYCHO_RAPL_DELAY_MS`	`0`	Expected delay between workload change and RAPL visibility (ms).
`TYCHO_GPU_DELAY_MS`	`200`	Expected delay between workload change and GPU visibility (ms).
`TYCHO_REDFISH_DELAY_MS`	`0`	Expected delay between workload change and Redfish visibility (ms).
`TYCHO_REDFISH_HEARTBEAT_MAX_GAP_MS`	`3000`	Maximum tolerated gap between consecutive Redfish samples (ms).
Autotuning controls		
`TYCHO_RAPL_POLL_AUTOTUNE`	`true`	Enables automatic calibration of RAPL polling interval.
`TYCHO_RAPL_DELAY_AUTOTUNE`	`true`	Enables automatic calibration of RAPL delay.
`TYCHO_GPU_POLL_AUTOTUNE`	`true`	Enables automatic calibration of GPU polling interval.
`TYCHO_GPU_DELAY_AUTOTUNE`	`true`	Enables automatic calibration of GPU delay.
`TYCHO_REDFISH_POLL_AUTOTUNE`	`true`	Enables automatic calibration of Redfish polling interval.
`TYCHO_REDFISH_DELAY_AUTOTUNE`	`true`	Enables automatic calibration of Redfish delay.
Analysis parameters		
`TYCHO_ANALYSIS_TRIGGER`	`"timer"`	Defines analysis trigger: `redfish` or `timer`.
`TYCHO_ANALYSIS_EVERY_SEC`	`15`	Interval for timer-based analysis (s).
`TYCHO_ANALYSIS_DETECT_LONGEST_DELAY`	`false`	Enables detection of the longest observed metric delay.

TABLE 6.6: User-facing configuration variables available in Tycho.

In summary, Tycho's configuration layer provides a single, immutable snapshot of all timing, polling, and analysis parameters. It is loaded once at startup, validated, normalized to a shared timebase, and propagated to all collectors and analysis modules. The design prioritizes reproducibility and correctness: no live reconfiguration is supported, and all behaviour is derived from explicit user input or well-defined defaults. This central configuration snapshot ensures that the timing engine, collectors, and buffering subsystem operate in a coherent, predictable manner throughout the entire runtime.

6.7 Timing Engine

6.7.1 Overview and Motivation

Tycho introduces a dedicated timing engine that replaces the synchronous update loop used in Kepler with an event-driven, per-metric scheduling layer. While the conceptual motivation for this change was discussed in § **??**, its practical purpose is

straightforward: to decouple the collection frequencies of heterogeneous telemetry sources and to establish a common temporal reference for subsequent analysis.

Each collector in Tycho (e.g., RAPL, eBPF, GPU, Redfish) operates under its own polling interval and is triggered by an aligned ticker maintained by the timing engine. All tickers share a single epoch (base timestamp) and are aligned to a configurable time quantum, ensuring deterministic phase relationships and bounded drift across all metrics. This architecture allows high-frequency sources to capture fine-grained temporal variation while preserving coherence with slower metrics.

The timing engine thus provides the temporal backbone of Tycho: it defines *when* each collector produces samples and ensures that all samples can later be correlated on a unified, monotonic timeline. Collected samples are pushed immediately into per-metric ring buffers, described in § 6.8, which retain recent histories for downstream integration and attribution.

6.7.2 Architecture and Design

The timing engine is implemented in the `engine.Manager` module. It acts as a lightweight scheduler that governs the execution of all metric collectors through independent, phase-aligned tickers. During initialization, each collector registers its callback function, polling interval, and enable flag with the manager. Once started, the manager creates one aligned ticker per enabled registration and launches each collector in a dedicated goroutine. All tickers share a single epoch, captured at startup, to guarantee deterministic alignment across collectors.

This design contrasts sharply with the global ticker used in Kepler, where a single update loop refreshed all metrics at a fixed interval. In Tycho, each ticker operates at its own cadence, determined by the configured polling period of the respective collector. For instance, RAPL may poll every 50 ms, GPU metrics every 200 ms, and Redfish telemetry every second, yet all remain phase-aligned through the shared epoch.

To maintain temporal consistency, the timing engine relies on the `clock` package, which defines both the aligned ticker and a monotonic timeline abstraction. The aligned ticker computes the initial delay to the next multiple of the polling period and then emits ticks at strictly periodic intervals. Each emitted epoch is converted into Tycho's internal time representation using the `Mono` clock, which maps wall-clock time to discrete quantum indices. The quantum defines the global temporal resolution (default: 1 ms) and guarantees strictly non-decreasing tick values, even under concurrency or system jitter.

The engine imposes minimal constraints on collector behavior: callbacks are expected to perform non-blocking work, typically pushing samples into the respective ring buffer, and to return immediately. This ensures low scheduling jitter and prevents slow collectors from influencing others. Lifecycle control is context-driven: when the execution context is cancelled, all ticker goroutines stop gracefully, and the manager waits for their completion before shutdown.

6.7.3 Synchronization and Collector Integration

All collectors in Tycho are synchronized through a shared temporal reference established at engine startup. The `Manager` captures a single epoch and provides it to every aligned ticker, ensuring that all collectors operate on the same epoch even if their polling intervals differ by several orders of magnitude. As a result, each collector's tick sequence can be expressed as a deterministic multiple of the global epoch, allowing later correlation between independently sampled metrics without interpolation artefacts.

Collectors register themselves before the timing engine is started. Each registration includes the collector's name, polling period, enable flag, and a `collect()` callback that executes whenever the corresponding ticker emits a tick. This callback receives both the current execution context and the aligned epoch, which is immediately converted into Tycho's internal monotonic time representation via the `Mono.From()` function. The collector then packages its raw measurements into a typed sample and pushes it to its corresponding ring buffer.

Because all collectors share the same monotonic clock and quantization step, the resulting sample streams can be merged and compared without further time normalization. Fast sources, such as RAPL or eBPF, provide dense sequences of measurements at fine granularity, while slower sources such as Redfish or GPU telemetry produce sparser but phase-aligned data points. This synchronization model eliminates the implicit coupling between sources that existed in Kepler and replaces it with a deterministic, time-driven coordination layer suitable for high-frequency, heterogeneous metrics.

6.7.4 Lifecycle and Configuration

The timing engine is initialized during Tycho's startup phase, after the metric collectors and buffer managers have been constructed. Before activation, each collector registers its collection parameters with the `Manager`, including polling intervals, enable flags, and callback references. Once registration is complete, the engine locks its configuration and starts the aligned tickers. Further modifications are prevented to guarantee a stable scheduling environment during runtime.

At startup, all timing parameters are validated and normalized. Invalid or negative values are rejected or normalized to safe defaults, and the global quantum is verified to be strictly positive. Polling intervals and buffer windows are cross-checked to ensure consistency across collectors, and derived values such as buffer sizes are recomputed from the validated configuration. This guarantees deterministic timing behavior even under partial or malformed configuration files.

The configuration layer also provides flexible control over measurement cadence. Polling periods for individual collectors can be adjusted independently, allowing users to balance temporal precision against system overhead. The default parameters represent a high-frequency but safe baseline: 50 ms for RAPL, 50 ms for eBPF, 200 ms for GPU, and 1 s for Redfish telemetry. All tickers are aligned to the global epoch defined by the monotonic clock, ensuring that these differences in cadence do not lead to drift over time.

Engine termination is context-driven: cancellation of the parent context signals all

tickers to stop, after which the manager waits for all goroutines to complete. This unified shutdown mechanism ensures a clean and deterministic teardown sequence without leaving residual workers or buffers in undefined states.

6.7.5 Discussion and Limitations

The timing engine establishes the foundation for Tycho's decoupled and fine-grained metric collection. By aligning all collectors to a shared epoch while allowing individual polling intervals, it eliminates the rigid synchronization that limited Kepler's temporal accuracy. This design provides a lightweight yet deterministic coordination layer, enabling heterogeneous telemetry sources to contribute time-consistent samples at their native cadence.

The engine's strengths lie in its simplicity and extensibility. Each collector operates independently, governed by its own aligned ticker, while context-driven lifecycle control ensures deterministic startup and shutdown. Because callbacks perform minimal, non-blocking work, jitter remains bounded even at high polling frequencies. This structure scales naturally with the number of collectors and provides a separation between timing logic, collection routines, and subsequent analysis stages.

Nevertheless, several practical limitations remain. The current implementation assumes a stable system clock and does not compensate for jitter introduced by the Go runtime or external scheduling delays. Collectors are expected to execute quickly; long-running or blocking operations may distort effective sampling intervals. Moreover, the engine's alignment is restricted to a single node and does not extend to multi-host synchronization, which would require external clock coordination. At very high sampling rates, the cumulative scheduling overhead may also become non-negligible on resource-constrained systems.

Despite these constraints, the timing engine represents a decisive architectural improvement over Kepler's fixed-interval model. It provides the temporal backbone for Tycho's data collection pipeline and enables accurate, high-resolution correlation across diverse telemetry sources. The following section, § 6.8, describes how these samples are buffered and retained for subsequent analysis, completing the temporal layer that underpins Tycho's measurement and attribution framework.

6.8 Ring Buffer Implementation

6.8.1 Overview

Tycho employs a per-metric ring buffer to store recent collection ticks produced by the individual collectors. Each collector owns a dedicated buffer that maintains a fixed number of entries, replacing the oldest values once full. This approach provides predictable memory usage and allows fast, allocation-free access to recent measurement histories. All ticks are stored in chronological order and include a monotonic epoch, ensuring consistent temporal alignment with the timing engine. The buffers are primarily used as transient storage for downstream analysis, enabling energy and utilization data to be correlated across metrics without incurring synchronization overhead.

6.8.2 Data Model and Sample Types

Each ring buffer is strongly typed and holds a single metric-specific tick structure. These tick types encapsulate all data collected during one polling interval and embed the `SampleMeta` structure, which records Tycho's monotonic epoch. Depending on the metric, a tick may contain simple scalar values (e.g., total node power) or collections of per-entity deltas (e.g., per-process counters, per-GPU readings, or per-domain energy data). For example, a `RaplTick` stores per-socket energy deltas across all domains, while a `BpfTick` aggregates process-level counters and hardware event deltas observed during that tick. This typed approach simplifies access and ensures that all metric data (regardless of complexity) can be correlated on a uniform temporal axis defined by the timing engine.

6.8.3 Dynamic Sizing and Spare Capacity

The capacity of each ring buffer is determined dynamically at startup from the configured buffer window and the polling interval of the corresponding collector. This calculation is performed by the `SizeForWindow()` function, which estimates the number of ticks required to represent the desired time window and adds a small margin of spare capacity to tolerate irregular sampling or short bursts of delayed polls. As a result, each buffer maintains a stable temporal horizon while avoiding premature overwrites during transient load variations. If configuration changes occur, buffers can be resized at runtime, preserving the most recent entries to ensure data continuity across reinitializations.

6.8.4 Thread Safety and Integration

Each ring buffer can be wrapped in a synchronized variant to ensure safe concurrent access between collectors and analysis routines. The synchronized type, `Sync[T]`, extends the basic ring with a read-write mutex, allowing simultaneous readers while protecting write operations during tick insertion. In practice, collectors append new ticks concurrently to their respective synchronized buffers, while downstream components such as the analysis engine or exporters read snapshots asynchronously. A central `Manager` maintains references to all buffers, handling creation, resizing, and typed access. This design provides deterministic retention and thread safety without introducing locking overhead into the collectors themselves, keeping the critical path lightweight and predictable.

6.9 Calibration

Calibration in Tycho serves two distinct purposes: determining suitable polling intervals for hardware interfaces with irregular publish behaviour, and quantifying the delay between workload onset and observable changes in a given metric. Calibration is applied only where hardware characteristics exhibit variability that materially affects temporal alignment or accuracy. Where metric sources are stable, low-latency, or analytically understood, Tycho uses fixed, justified settings instead of dynamic calibration.

In practice, Tycho performs calibration for a small subset of collectors:

- **Polling-frequency calibration** is required only for GPU and Redfish power metrics, whose publish cadence is hardware-controlled and irregular. eBPF

metrics are event-driven and do not require calibration, and RAPL counters update at sub-millisecond granularity with well-characterised behaviour, making calibration unnecessary.

- **Delay calibration** is performed exclusively for GPU power metrics, where internal averaging and buffered updates introduce measurable reaction latency. RAPL and eBPF have negligible or analytically bounded delay at Tycho's sampling scale, and Redfish delay is too irregular and too coarse for meaningful calibration.

All remaining collectors operate with fixed, analysis-driven parameters. The following subsections describe Tycho's calibration strategy and procedures in detail.

6.9.1 Polling-frequency Calibration

Polling-frequency calibration identifies a suitable sampling interval for sources whose publish cadence is neither fixed nor documented. For some collectors (e.g. RAPL, eBPF), the hardware already provides reliable or sufficiently fast update behaviour, making calibration unnecessary. For others (GPU, Redfish), Tycho uses brief hyperpolling phases to infer the effective update rhythm before normal collection begins.

6.9.1.1 eBPF Polling frequency calibration

Tycho does not perform a dedicated polling-frequency calibration for eBPF-based utilization metrics. eBPF events can be sampled at arbitrarily high rates, and their timing does not depend on hardware update intervals. Instead, Tycho aligns eBPF sampling with the RAPL polling frequency to maintain temporal consistency across collectors. Allowing shorter eBPF intervals would provide limited additional benefit while increasing processing overhead, so eBPF adopts the same lower bound as the CPU energy path and does not require separate calibration.

6.9.1.2 RAPL Polling frequency calibration

RAPL updates energy counters at sub-millisecond granularity, but sampling too aggressively introduces noise and diminishes measurement quality. Accuracy is further affected by optional energy filtering mechanisms that reduce fine-grained observability[26, Table 2-2]. Jay et al. found that sampling slower than 50 Hz keeps relative error below 0.5%[24]. Tycho imposes a minimum RAPL polling interval of 50 ms and treats any faster sampling as unnecessary. Because this bound is derived from hardware behaviour rather than runtime variability, Tycho does not include a polling-frequency calibration mechanism for RAPL.

6.9.1.3 GPU Polling frequency calibration

Before enabling regular GPU collection, Tycho measures the effective publish cadence of NVML power metrics. Rather than searching over candidate periods, it uses a short *hyperpoll* phase: for the duration of the calibration window, Tycho queries each GPU at a fixed, conservatively fast interval and simply counts how often it sees a new, valid NVML update. From these timestamps it derives inter-arrival gaps and summarises them (via the median) into an estimated publish interval per device.

This converts the problem into "hits over time": if a GPU produces n distinct updates over a calibration window of length T, the observed publish cadence is approximately T/n, refined by looking at the distribution of individual gaps rather than a single average. Tycho then recommends a per-device polling interval based on this estimate and finally adopts the most conservative (fastest) setting across all GPUs as the node-wide GPU polling period. This approach keeps the implementation simple while ensuring that subsequent GPU collection runs fast enough to see every NVML update without imposing unnecessary overhead.

6.9.1.4 Redfish Polling frequency calibration

Redfish power readings are coarse and highly irregular. Publish intervals may vary from sub-second to multi-second gaps, and this variability is further amplified on BMCs that expose multiple chassis or subsystems. Tycho therefore applies a simplified calibration procedure similar to the GPU polling calibration, but adapted to the characteristics of Redfish.

During calibration, Tycho hyperpolls Redfish at the minimum polling interval (500 ms) for a short window (60 seconds). Every newly observed Redfish update (from any chassis exposed by the same BMC) contributes an inter-arrival gap. Using the median of these gaps provides a robust estimate of the typical publish interval while naturally reflecting multi-chassis setups: if one chassis updates faster than others, the calibration converges to that faster cadence, ensuring that no subsystem is undersampled.

Based on this estimate, Tycho selects an operational polling period:

- Without continuous heartbeat, the median publish interval (clamped to 500 ms) is used directly.
- With continuous heartbeat enabled, Tycho selects a smaller polling interval (alf the median), allowing the collector to detect new samples promptly and maintain accurate freshness tracking despite Redfish's inherent timing jitter.

This yields a coarse but sufficient estimate of the underlying BMC cadence while relying on the adaptive heartbeat mechanism during normal operation to maintain temporal coherence across all chassis.

6.9.2 Delay Calibration

Delay calibration determines the time interval between the start of a workload and the first measurable reaction in the corresponding hardware metric. Because Tycho itself does not execute workloads on the host and does not have direct access to specialised hardware resources, delay calibration must be performed by external scripts that run on the bare-metal node. Each calibration attempt begins with an idle period until the metric reaches a stationary state, followed by a controlled workload with a known start time. The delay is the earliest sample that exceeds the idle baseline by a detectable margin. Since many hardware metrics apply internal averaging or have irregular publish cycles, a single run is not sufficient. Multiple runs are required to obtain a stable distribution. Tycho uses either the minimum or the fifth percentile of observed delays. The minimum reflects the earliest possible reaction. The fifth percentile can be used when the minimum appears to be an outlier.

6.9.2.1 GPU delay calibration

GPU delay calibration uses a dedicated script that generates a controlled GPU workload and monitors NVML power readings. A preliminary attempt relied on gpu-burn[59], but this tool carries a non-negligible startup delay that obscures the true hardware reaction time. To address this, the calibration mechanism was reimplemented with Numba[60], which allows the script to launch a custom floating-point kernel with exact control over the workload timing.

The script alternates between idle and active periods. During idle periods, NVML power is sampled until the readings reach a stable baseline, and only the final portion of the idle window is used for statistical analysis. During active periods, the Numba kernel saturates the GPU's compute units while the script continually samples NVML power. The delay is identified as the first sample that exceeds the idle baseline by a small, adaptively computed threshold. Because NVML power reports are averaged over approximately one second, many runs are required to gather a suitable distribution of delays. The default configuration uses 15 seconds of idle time, 15 seconds of active workload, a sampling interval of 50 milliseconds, and 100 runs.

6.9.2.2 RAPL delay calibration

No dedicated delay calibration is planned for RAPL. With the fast update frequency of RAPL energy counters, access latency is negligible. Although very early work criticised timing characteristics at sub-millisecond resolution[23], later studies generally consider RAPL accurate for the time scales relevant to energy modelling. Tycho enforces a minimum collection interval of 50 milliseconds because RAPL readings are noisy at very short intervals[21]. At this granularity any residual delay is small compared to the sampling window and does not meaningfully affect alignment. Explicit delay calibration would therefore provide minimal benefit.

6.9.2.3 Redfish delay calibration

Redfish presents a fundamentally different problem. Power readings are published slowly, with irregular inter-arrival times, and may skip updates entirely. An earlier empirical study reported delays of roughly 200 milliseconds[4], but also noted substantial variability across systems. Additional indeterminism arises from the network path and the unknown internal behaviour of the BMC. Since Tycho already mitigates staleness through its freshness mechanism, explicit delay calibration is neither feasible nor useful. Redfish is therefore treated as a coarse, low-resolution metric, appropriate for slow global trends but not for fine-grained timing.

6.9.2.4 eBPF Metrics delay calibration

eBPF-based utilisation metrics behave differently. They are collected directly in kernel context and do not involve additional publish intervals or device-side buffering. Their effective delay is negligible relative to Tycho's sampling windows, so no delay calibration is required.

6.10 Metadata Subsystem

Tycho introduces a dedicated metadata subsystem that provides an accurate, temporally aligned view of the node's execution state. It follows the general Tycho design principles introduced in § **??**: separation of concerns, accuracy-first data selection, and consistent correlation via monotonic timestamps. In contrast to *Kepler*, where metadata handling is tightly coupled to the individual metric collectors and suffers from the limitations discussed in § **??** and § **??**, Tycho treats metadata as an independent architectural layer with its own storage and timing model.

6.10.1 Scope of Metadata

The subsystem aggregates all information required for correct and high-fidelity attribution. Rather than collecting only the minimal subset, Tycho records all metadata that materially improves attribution accuracy or interpretability. This includes:

- Process identity and attributes: PID, command name, cgroup, start time, and container association.
- Container and pod identity: container ID, container name, pod name, namespace, and basic lifecycle state.
- Kubelet-derived status: running, terminating, completed, or evicted pods and containers.
- Optional cAdvisor metadata: CPU, memory, and IO-level container usage counters for cross-validation and filtering.

Each of these sources contributes a partial view; the metadata subsystem maintains a unified, time-aligned representation by merging them into a shared store.

6.10.2 Positioning Within Tycho

Metadata collection is fully decoupled from power and utilization sampling. Collectors run on lightweight wall-clock intervals and push updates into a central store, while the analysis layer retrieves all required metadata when performing attribution. This prevents temporal entanglement with high-frequency collectors and avoids the failure modes observed in prior systems where stale container or pod state leaked into energy attribution.

6.10.3 Metadata Store and Lifetime Management

All metadata produced by Tycho's collectors is written into a shared in-memory store that represents the node's recent execution context. The store maintains three independent maps for processes, containers, and pods, each keyed by a stable identifier. Every entry is annotated with two timestamps: a monotonic timestamp for correlation with power and utilization samples, and a wall-clock timestamp used for enforcing the same horizon that governs Tycho's power and utilization buffers.

The store maintains no long-term history. Instead, it retains a short rolling window of entries whose timestamps fall within the same horizon used by Tycho's power and utilization buffers; the metadata retention period is therefore exactly the configured `maxAge`. Within this window, the analysis layer can reconstruct a coherent,

time-aligned view of processes, containers, and pods for any attribution interval. Collectors only insert or update metadata; they never delete entries directly.

Removal is delegated entirely to a horizon-based garbage collector. During each garbage-collection pass, the store removes entries whose wall-clock timestamp is older than the configured horizon. This ensures that terminated or deleted entities remain visible long enough for overlapping attribution windows to resolve correctly, while preventing stale metadata from influencing later analysis. When collectors stop producing updates, the store drains itself naturally after the horizon expires. The result is deterministic freshness guarantees, bounded memory usage, and a simple, robust metadata lifecycle.

The following subsections describe the individual metadata collectors that populate this store.

6.10.4 Process Metadata Collector

The process metadata collector maintains a short-horizon view of all processes running on the node. Its purpose is to provide the analysis layer with enough contextual information to correlate per-process activity with container and pod identities, while avoiding any direct dependency on power or utilization collectors.

The collector performs a best-effort enumeration of all processes via `/proc`. For each PID it records a minimal set of attributes useful for later energy attribution: a stable per-boot identifier (`PID`, `StartJiffies`), a container mapping, and a human-readable command name. Metadata is timestamped with monotonic and wall-clock time and inserted into the central metadata store, which enforces a sliding time horizon through periodic garbage collection.

Reused functionality from Kepler Tycho reuses Kepler's cgroup resolution logic to map processes to container identifiers. This logic extracts normalized container IDs from cgroup paths and distinguishes pod containers from system processes. Tycho integrates this component without modifying its behaviour.

Tycho-specific additions Tycho introduces a new, self-contained metadata subsystem and defines the process collector as an independent, low-overhead component. In contrast to Kepler, Tycho does not combine process enumeration with resource accounting. The collector records a stable process start token (`StartJiffies`), used only to disambiguate PID reuse, and stores all metadata in a dedicated in-memory store shared across all metadata collectors. No attribution logic is implemented at this stage.

The process collector intentionally keeps its scope minimal. Higher-level enrichment such as pod metadata, QoS class or container state is delegated to the kubelet and cgroup-based container collectors described in § 6.10.5.

6.10.4.1 Collected Metrics

The following table 6.7 shows the collected metrics.

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 6.7: Process metadata collected by the process collector

6.10.5 Kubelet Metadata Collector

The kubelet metadata collector provides Tycho with an authoritative, scheduler-consistent view of all pods and containers currently known to the node. It complements the process collector by supplying the semantic information required for correct attribution: pod identity, lifecycle state, container status, and resource specifications. Unlike the process collector, which observes execution from the operating system perspective, the kubelet collector captures the Kubernetes control-plane perspective.

The collector periodically retrieves the full pod list from the kubelet's `/pods` endpoint and extracts only metadata that cannot be reconstructed later. All entries are timestamped with Tycho's monotonic timebase and inserted into the shared metadata store, where they remain available for attribution until removed by horizon-based garbage collection.

Reused functionality from Kepler Tycho reuses Kepler's container-ID normalization logic to extract runtime container identifiers from kubelet status fields. The use of the kubelet's PodStatus API as the ground truth for container lifecycle state is also conceptually inherited from Kepler, but Tycho decouples it from resource accounting and avoids Kepler's tightly coupled watcher design.

Tycho-specific additions Tycho substantially extends the kubelet collector relative to Kepler:

- Pod-level and container-level metadata are stored in a dedicated subsystem with monotonic timestamps, ensuring temporal alignment with power and utilization sampling.
- Resource requests and limits from `pod.spec` are recorded for both containers and aggregated pods, preserving information that disappears once pods terminate.

- Controller owner references (e.g. ReplicaSet, DaemonSet) are captured to support later grouping and attribution without encoding any classification logic in the collector.
- Ephemeral and init containers are fully supported, and termination state and exit codes are recorded to prevent attribution to completed workloads. Terminated containers remain in the store until the horizon expires, ensuring correct attribution for analysis windows that overlap their termination.

No scheduling decisions, classification, or attribution logic is executed at collection time; all interpretation is deferred to the analysis layer.

6.10.5.1 Collected Metrics

The kubelet collector records per-pod and per-container metadata as shown in Tables 6.8 and 6.9. Only fields that cannot be reliably reconstructed later are persisted.

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 6.8: 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 6.9: Container metadata collected by the kubelet collector

6.10.6 Why Tycho Does Not Require cAdvisor for Container Enumeration

KubeWatt integrates *cAdvisor* to correct several metadata inconsistencies observed in Kepler, in particular the presence of slice-level cgroups and terminated containers that continued to appear in resource usage reports. In KubeWatt's design, cAdvisor serves primarily as a *filter*: it exposes only real, runtime-managed containers and thereby removes artefacts such as `kubepods.slice` or QoS slices that would otherwise distort CPU attribution [52].

Tycho does not require this mechanism. The root causes that motivated the use of cAdvisor in KubeWatt are addressed structurally by Tycho's metadata subsystem: container and pod identity are obtained exclusively from the kubelet, process–container association is resolved using a stable cgroup parser, and all metadata is managed within a unified, timestamped store with deterministic garbage collection. As a result, Tycho never encounters the slice-level or terminated-container artefacts that cAdvisor was used to filter out.

This does not preclude using cAdvisor as an optional enrichment source for slowly changing contextual metrics (for example throttling counters or memory usage), but its role as a correctness or filtering layer is superseded by Tycho's dedicated metadata architecture.

Bibliography

[1] Caspar Wackerle. *PowerStack: Automated Kubernetes Deployment for Energy Efficiency Analysis*. GitHub repository. 2025. URL: https://github.com/casparwackerle/PowerStack.

[2] Weiwei Lin et al. "A Taxonomy and Survey of Power Models and Power Modeling for Cloud Servers". In: *ACM Comput. Surv.* 53.5 (Sept. 2020), 100:1–100:41. ISSN: 0360-0300. DOI: 10.1145/3406208. (Visited on 04/20/2025).

[3] Saiqin Long et al. "A Review of Energy Efficiency Evaluation Technologies in Cloud Data Centers". In: *Energy and Buildings* 260 (Apr. 2022), p. 111848. ISSN: 0378-7788. DOI: 10.1016/j.enbuild.2022.111848. (Visited on 04/20/2025).

[4] Yewan Wang et al. "An Empirical Study of Power Characterization Approaches for Servers". In: *ENERGY 2019 - The Ninth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*. June 2019, p. 1. (Visited on 04/23/2025).

[5] Charles Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. New York, NY, USA: Association for Computing Machinery, Oct. 2012, pp. 1–13. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391236. (Visited on 11/26/2025).

[6] Muhammad Waseem et al. *Containerization in Multi-Cloud Environment: Roles, Strategies, Challenges, and Solutions for Effective Implementation*. July 2025. DOI: 10.48550/arXiv.2403.12980. arXiv: 2403.12980 [cs]. (Visited on 11/26/2025).

[7] Emiliano Casalicchio and Stefano Iannucci. "The State-of-the-Art in Container Technologies: Application, Orchestration and Security". In: *Concurrency and Computation: Practice and Experience* 32.17 (2020), e5668. ISSN: 1532-0634. DOI: 10.1002/cpe.5668. (Visited on 11/26/2025).

[8] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. "A Validation of DRAM RAPL Power Measurements". In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 455–470. DOI: 10.1145/2989081.2989088. (Visited on 05/21/2025).

[9] Steven van der Vlugt et al. *PowerSensor3: A Fast and Accurate Open Source Power Measurement Tool*. Apr. 2025. DOI: 10.48550/arXiv.2504.17883. arXiv: 2504.17883 [cs]. (Visited on 05/09/2025).

[10] UEFI Forum. *Advanced Configuration and Power Interface Specification Version 6.6*. Accessed April 2025. Sept. 2021. URL: https://uefi.org/sites/default/files/resources/ACPI_Spec_6.6.pdf.

[11] Richard Kavanagh, Django Armstrong, and Karim Djemame. "Accuracy of Energy Model Calibration with IPMI". In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. June 2016, pp. 648–655. DOI: 10.1109/CLOUD.2016.0091. (Visited on 04/23/2025).

[12] Richard Kavanagh and Karim Djemame. "Rapid and Accurate Energy Models through Calibration with IPMI and RAPL". In: *Concurrency and Computation: Practice and Experience* 31.13 (2019), e5124. ISSN: 1532-0634. DOI: 10.1002/cpe.5124. (Visited on 04/23/2025).

[13] Magnus Herrlin. "Accessing Onboard Server Sensors for Energy Efficiency in Data Centers". In: (Sept. 2021). (Visited on 11/27/2025).

[14] Ghazanfar Ali et al. "Redfish-Nagios: A Scalable Out-of-Band Data Center Monitoring Framework Based on Redfish Telemetry Model". In: *Fifth International Workshop on Systems and Network Telemetry and Analytics*. Minneapolis MN USA: ACM, June 2022, pp. 3–11. ISBN: 978-1-4503-9315-7. DOI: 10.1145/3526064.3534108. (Visited on 11/27/2025).

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 3B, Chapter 16.10: Plaform Specific Power Management Support. Available online: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. 2024.

[16] Guillaume Raffin and Denis Trystram. "Dissecting the Software-Based Measurement of CPU Energy Consumption: A Comparative Analysis". In: *IEEE Transactions on Parallel and Distributed Systems* 36.1 (Jan. 2025), pp. 96–107. ISSN: 1558-2183. DOI: 10.1109/TPDS.2024.3492336. (Visited on 04/02/2025).

[17] Daniel Hackenberg et al. "An Energy Efficiency Feature Survey of the Intel Haswell Processor". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. May 2015, pp. 896–904. DOI: 10.1109/IPDPSW.2015.70. (Visited on 04/28/2025).

[18] Daniel Hackenberg et al. "Power Measurement Techniques on Standard Compute Nodes: A Quantitative Comparison". In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2013, pp. 194–204. DOI: 10.1109/ISPASS.2013.6557170. (Visited on 04/28/2025).

[19] Lukas Alt et al. "An Experimental Setup to Evaluate RAPL Energy Counters for Heterogeneous Memory". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. ICPE '24. New York, NY, USA: Association for Computing Machinery, May 2024, pp. 71–82. ISBN: 979-8-4007-0444-4. DOI: 10.1145/3629526.3645052. (Visited on 04/02/2025).

[20] Tom Kennes. *Measuring IT Carbon Footprint: What Is the Current Status Actually?* June 2023. DOI: 10.48550/arXiv.2306.10049. arXiv: 2306.10049 [cs]. (Visited on 04/23/2025).

[21] Robert Schöne et al. "Energy Efficiency Features of the Intel Alder Lake Architecture". In: *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*. London United Kingdom: ACM, May 2024, pp. 95–106. ISBN: 979-8-4007-0444-4. DOI: 10.1145/3629526.3645040. (Visited on 04/07/2025).

[22] Robert Schöne et al. "Energy Efficiency Aspects of the AMD Zen 2 Architecture". In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2021, pp. 562–571. DOI: 10.1109/Cluster48925.2021.00087. (Visited on 04/28/2025).

[23] Kashif Nizam Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (Mar. 2018), 9:1–9:26. ISSN: 2376-3639. DOI: 10.1145/3177754. (Visited on 04/07/2025).

[24] Mathilde Jay et al. "An Experimental Comparison of Software-Based Power Meters: Focus on CPU and GPU". In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. May 2023, pp. 106–118. DOI: 10.1109/CCGrid57682.2023.00020. (Visited on 04/21/2025).

[25] Moritz Lipp et al. "PLATYPUS: Software-based Power Side-Channel Attacks on X86". In: *2021 IEEE Symposium on Security and Privacy (SP)*. May 2021, pp. 355–371. DOI: 10.1109/SP40001.2021.00063. (Visited on 05/21/2025).

[26] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers*. Tech. rep. 335592-081US. Accessed 2025-04-28. Intel Corporation, Sept. 2023. URL: https://cdrdv2.intel.com/v1/dl/getContent/671098.

[27] Zeyu Yang, Karel Adamek, and Wesley Armour. "Accurate and Convenient Energy Measurements for GPUs: A Detailed Study of NVIDIA GPU's Built-In Power Sensor". In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2024, pp. 1–17. DOI: 10.1109/SC41406.2024.00028. (Visited on 05/09/2025).

[28] Oscar Hernandez et al. "Preliminary Study on Fine-Grained Power and Energy Measurements on Grace Hopper GH200 with Open-Source Performance Tools". In: *Proceedings of the 2025 International Conference on High Performance Computing in Asia-Pacific Region Workshops*. Hsinchu Taiwan: ACM, Feb. 2025, pp. 11–22. ISBN: 979-8-4007-1342-2. DOI: 10.1145/3703001.3724383. (Visited on 11/27/2025).

[29] Le Mai Weakley et al. "Monitoring and Characterizing GPU Usage". In: *Concurrency and Computation: Practice and Experience* 37.3 (2025), e8341. ISSN: 1532-0634. DOI: 10.1002/cpe.8341. (Visited on 11/27/2025).

[30] The Linux Kernel Community. *The proc Filesystem*. https://www.kernel.org/doc/html/latest/filesystems/proc.html. Accessed: 2025-06-17. 2025.

[31] The Linux Kernel Community. *Control Group v1 — Linux Kernel Documentation*. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/index.html. Accessed: 2025-06-17. 2025.

[32] The Linux Kernel Community. *Control Group v2 — Linux Kernel Documentation*. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html. Accessed: 2025-06-17. 2025.

[33] Cilium Authors. *eBPF and XDP Reference Guide*. https://docs.cilium.io/en/latest/reference-guides/bpf/index.html. Accessed: 2025-06-17. 2025.

[34] Cyril Cassagnes et al. "The Rise of eBPF for Non-Intrusive Performance Monitoring". In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. Apr. 2020, pp. 1–7. DOI: 10.1109/NOMS47738.2020.9110434. (Visited on 06/14/2025).

[35] Brendan Gregg. *CPU Utilization is Wrong*. Blog post. Accessed 29 June 2025. May 2017. URL: https://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html.

[36] smartmontools developers. *smartmontools: Control and monitor storage systems using S.M.A.R.T.* https://github.com/smartmontools/smartmontools/. Accessed May 2025. 2025.

[37] Linux NVMe Maintainers. *nvme-cli: NVMe management command line interface*. https://github.com/linux-nvme/nvme-cli. Accessed May 2025. 2025.

[38] Seokhei Cho et al. "Design Tradeoffs of SSDs: From Energy Consumption's Perspective". In: *ACM Trans. Storage* 11.2 (Mar. 2015), 8:1–8:24. ISSN: 1553-3077. DOI: 10.1145/2644818. (Visited on 05/18/2025).

[39] Yan Li and Darrell D.E. Long. "Which Storage Device Is the Greenest? Modeling the Energy Cost of I/O Workloads". In: *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*. Sept. 2014, pp. 100–105. DOI: 10.1109/MASCOTS.2014.20. (Visited on 05/19/2025).

[40] Eric Borba, Eduardo Tavares, and Paulo Maciel. "A Modeling Approach for Estimating Performance and Energy Consumption of Storage Systems". In: *Journal of Computer and System Sciences* 128 (Sept. 2022), pp. 86–106. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2022.04.001. (Visited on 05/18/2025).

[41] Ripduman Sohan et al. "Characterizing 10 Gbps Network Interface Energy Consumption". In: *IEEE Local Computer Network Conference*. Oct. 2010, pp. 268–271. DOI: 10.1109/LCN.2010.5735719. (Visited on 05/30/2025).

[42] Robert Basmadjian et al. "Cloud Computing and Its Interest in Saving Energy: The Use Case of a Private Cloud". In: *Journal of Cloud Computing: Advances, Systems and Applications* 1.1 (June 2012), p. 5. ISSN: 2192-113X. DOI: 10.1186/2192-113X-1-5. (Visited on 06/01/2025).

[43] Saeedeh Baneshi et al. "Analyzing Per-Application Energy Consumption in a Multi-Application Computing Continuum". In: *2024 9th International Conference on Fog and Mobile Edge Computing (FMEC)*. Sept. 2024, pp. 30–37. DOI: 10.1109/FMEC62297.2024.10710253. (Visited on 05/30/2025).

[44] TechNotes. *Deciphering the PCI Power States*. Accessed June 2025. Feb. 2024. URL: https://technotes.blog/2024/02/04/deciphering-the-pci-power-states/.

[45] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. "Power Provisioning for a Warehouse-Sized Computer". In: *SIGARCH Comput. Archit. News* 35.2 (June 2007), pp. 13–23. ISSN: 0163-5964. DOI: 10.1145/1273440.1250665. (Visited on 05/21/2025).

[46] Chung-Hsing Hsu and Stephen W. Poole. "Power Signature Analysis of the SPECpower_ssj2008 Benchmark". In: *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. Apr. 2011, pp. 227–236. DOI: 10.1109/ISPASS.2011.5762739. (Visited on 05/21/2025).

[47] Shuaiwen Leon Song, Kevin Barker, and Darren Kerbyson. "Unified Performance and Power Modeling of Scientific Workloads". In: *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*. E2SC '13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 1–8. ISBN: 978-1-4503-2504-2. DOI: 10.1145/2536430.2536435. (Visited on 05/21/2025).

[48] Jordi Arjona Aroca et al. "A Measurement-Based Analysis of the Energy Consumption of Data Center Servers". In: *Proceedings of the 5th International Conference on Future Energy Systems*. E-Energy '14. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 63–74. ISBN: 978-1-4503-2819-7. DOI: 10.1145/2602044.2602061. (Visited on 06/01/2025).

[49] Inc. Meta Platforms. *Kepler v0.9.0 (pre-rewrite): Kubernetes-based power and energy estimation framework*. Accessed: 2025-04-28. 2023. URL: https://https://github.com/sustainable-computing-io/kepler/releases/tag/v0.9.0.

[50] Bjorn Pijnacker. "Estimating Container-level Power Usage in Kubernetes". MA thesis. University of Groningen, Nov. 2024. (Visited on 03/17/2025).

[51] Linux Foundation Energy and Performance Working Group. *Kepler: Kubernetes-based Power and Energy Estimation Framework*. Accessed: 2025-11-14. 2025. URL: https://github.com/sustainable-computing-io/kepler.

[52] Bjorn Pijnacker, Brian Setz, and Vasilios Andrikopoulos. *Container-Level Energy Observability in Kubernetes Clusters*. Apr. 2025. DOI: 10.48550/arXiv.2504.10702. arXiv: 2504.10702 [cs]. (Visited on 07/02/2025).

[53] Hubblo-org. *Scaphandre Documentation*. Accessed: 2025-04-28. 2024. URL: https://github.com/hubblo-org/scaphandre-documentation.

[54] Guillaume Fieni, Romain Rouvoy, and Lionel Seinturier. "SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. May 2020, pp. 479–488. DOI: 10.1109/CCGrid49817.2020.00-45. (Visited on 05/21/2025).

[55] MLCO2. *CodeCarbon: Track emissions from your computing*. Accessed: 2025-04-28. 2023. URL: https://github.com/mlco2/codecarbon.

[56] Caspar Wackerle. *Tycho: an accuracy-first container-level energy consumption exporter for Kubernetes (based on Kepler v0.9)*. GitHub repository. 2025. URL: https://github.com/casparwackerle/tycho-energy.

[57] NVIDIA Corporation. *NVML API Reference Guide*. Function `nvmlDeviceGetPowerUsage`: retrieves GPU power usage in milliwatts. 2024. URL: https://docs.nvidia.com/deploy/pdf/NVML_API_Reference_Guide.pdf (visited on 11/13/2025).

[58] Yole Group. *Data Center Semiconductor Trends 2025: Artificial Intelligence Reshapes Compute and Memory Markets*. Press Release. 2025. URL: https://www.yolegroup.com/press-release/data-center-semiconductor-trends-2025-artificial-intelligence-reshapes-compute-and-memory-markets/.

[59] wilicc. *gpu-burn: Multi-GPU CUDA stress test*. https://github.com/wilicc/gpu-burn. Accessed: 2025-11-18. 2025.

[60] The Numba Developers. *Numba: A High Performance Python Compiler*. https://numba.pydata.org/. Accessed: 2025-11-18. 2025.