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
Copyright: Zurich University of Applied Sciences

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

1.1 Motivation

Global data center energy consumption is increasing rapidly, driven by the rising demand for compute-heavy workloads such as artificial intelligence and large-scale analytics. Data centers consumed roughly 1.5% of global electricity in 2024 (about 415 TWh) and are expected to more than double their consumption by 2030 [2]. At the same time, traditional sources of efficiency improvement, such as Moore's law and Dennard scaling, are slowing down [3, 4], while gains from infrastructure optimizations approach diminishing returns [5, 6].

Containerized workloads form an increasingly large share of this growing footprint. They provide lightweight and scalable deployment mechanisms [7], and Kubernetes has become the dominant platform for orchestrating such workloads at scale. Despite their operational benefits and generally low overhead [8], containers make energy transparency difficult. Their shared-resource architecture and multi-layer abstraction obscure the relationship between individual workloads and the physical energy they consume, which complicates research on energy efficiency. As interest in energy-aware computing grows, accurate and fine-grained measurement becomes essential for research, performance engineering, and scheduling. Existing tools offer useful approximations but rely heavily on heuristic models and asynchronous telemetry, which limits their validity and reproducibility.

Kepler[9], a CNCF-backed project under active development, has become a widely adopted energy observability tool for Kubernetes clusters. Its design prioritizes portability, low overhead, and safe deployment across diverse environments, which makes it highly suitable for operational use at scale. A detailed analysis of its methodology and design philosophy is provided in Appendix A. In this context, Tycho is not intended to compete with Kepler or replace it. Instead, Tycho explores a complementary space: the upper bound of what measurement accuracy is achievable when higher-privilege monitoring methods, synchronized server-wide telemetry, and more fine-grained event-time alignment are permitted. While such techniques are often impractical for general-purpose deployments, they offer substantial value for research settings and large-scale R&D environments that require precise and reproducible energy measurements. Tycho is an attempt to fill this accuracy-focused niche by implementing methods that prioritize measurement fidelity over broad deployability.

1.2 Challenges and Limitations of Existing Approaches

Current approaches to workload-level energy estimation provide useful insights but remain limited in accuracy and reproducibility. Tools such as Kepler and Scaphandre[10] correlate resource utilization metrics with node-level energy telemetry, yet they operate under constraints that make fine-grained attribution difficult. Many systems depend on asynchronous and low-frequency sampling, rely on heterogeneous telemetry sources with differing clock domains, or employ heuristic models that introduce uncertainty. Purpose-built, high-accuracy measurement hardware exists, but its cost and operational overhead prevent widespread adoption in production environments. As a result, most available solutions offer high-level trends rather than precise, time-aligned measurements required for rigorous analysis. A more detailed discussion of these limitations is provided in § **??** and further expanded in Appendix A.

1.3 Problem Context

Modern container platforms such as Kubernetes run large numbers of heterogeneous, short-lived workloads on shared physical infrastructure. In multi-tenant clusters, containers continuously start, stop, migrate, and compete for CPU, memory, storage, and I/O resources. This dynamism complicates any attempt to determine how much energy a specific workload is responsible for.

At the hardware level, telemetry relevant to energy measurement is fragmented across multiple subsystems. Interfaces such as Intel RAPL, Redfish power sensors, GPU telemetry, and eBPF-based utilization metrics differ in sampling frequency, latency, and clock domain. Their values may arrive asynchronously and with inconsistent timing, making direct alignment difficult. Reliable attribution requires correlating these subsystems to a common time base, which existing platforms do not provide.

Kubernetes adds another layer of complexity. Processes are encapsulated inside containers, which in turn map to pods, cgroups, and namespaces that evolve over time. Accurate attribution therefore depends on robust tracking of process-to-container relationships and on resolving the timing mismatch between telemetry sources and the rapid state changes within the cluster. These combined factors make precise, container-level energy attribution a challenging problem.

1.4 Problem Statement

1.5 Goals of This Thesis

1.6 Research Questions

1.7 Contributions

1.8 Positioning Within Previous Work

This thesis builds directly on two earlier specialization projects completed as part of the Master’s program. The first project (VT1) focused on the practical deployment of a bare-metal Kubernetes environment for energy efficiency research. It provided a fully automated setup procedure for cluster installation, persistent storage, monitoring infrastructure, and stress-test workloads, including all required configuration and connectivity. VT1 initially deployed Kepler as its energy monitoring component but has since been adapted to install Tycho instead, enabling rapid provisioning of complete measurement and evaluation environments. The resulting *PowerStack* repository [1] works in conjunction with this thesis by providing the operational foundation on which Tycho can be deployed and evaluated, and is documented in Appendix B.

The second project (VT2) examined the theoretical foundations of energy measurement in modern servers and containerized environments. It analyzed hardware-level telemetry interfaces, attribution challenges, and the strengths and limitations of existing tools, providing a structured overview of the state of the art. VT2 identified the methodological gaps that hinder precise workload-level energy attribution, many of which motivate the design choices in Tycho. A detailed version of this analysis is included in Appendix A.

The present thesis consolidates these two lines of work. It integrates the practical insights from VT1 with the theoretical findings of VT2 and develops Tycho as an accuracy-first system for server-wide energy measurement and workload attribution. In this role, the thesis serves as the point where the implementation, methodological foundations, and design philosophy come together into a coherent system.

1.9 Scope and Boundaries

1.10 Target Audience

1.11 Origin of the Name “Tycho”

1.12 Methodological Approach

1.13 Ethical and Sustainability Considerations

1.14 Thesis Structure

1.15 System Environment for Development, Build and Debugging

This section documents the environment used to develop, build, and debug *Tycho*; detailed guides live in [11].

1.15.1 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.

1.15.2 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*).

1.15.2.1 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.

1.15.2.2 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.

1.15.2.3 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.

1.15.2.4 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.

1.15.3 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.

1.15.3.1 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.

1.15.3.2 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.

1.15.3.3 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.

1.15.4 Supporting Tools and Utilities

1.15.4.1 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 § 1.15.2. Repository scripts map configuration keys to explicit flags for local runs, debug sessions, and ad hoc deploys.

1.15.4.2 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.

1.15.5 Relevance and Limitations

1.15.5.1 Scope and contribution

The development, build, and debugging environment described in § 1.15.2 and § 1.15.3 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 § 1.15 provide the operational detail that is intentionally omitted from the main text.

1.15.5.2 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 § 1.15.

1.16 eBPF-collector-Based CPU Time Attribution

1.16.1 Scope and Motivation

The kernel-level `eBPF` subsystem in Tycho provides the foundation for process-level energy attribution. It captures CPU scheduling, interrupt, and performance-counter events directly inside the Linux kernel, translating them into continuous measurements of CPU ownership and activity. All higher-level aggregation and modeling occur in userspace; this section therefore focuses exclusively on the in-kernel instrumentation and the data it exposes.

Kepler's original `eBPF` design offered a coarse but functional basis for collecting CPU time and basic performance metrics. Its `sched_switch` tracepoint recorded process runtime, while hardware performance counters supplied instruction and cache data. However, the sampling cadence and aggregation logic were controlled from userspace, producing irregular collection intervals and temporal misalignment with energy readings. Kepler also treated all CPU time as a single undifferentiated category, omitting explicit representation of idle periods, interrupt handling, and kernel threads. As a result, a portion of the processor's activity (often significant under I/O-heavy workloads) remained unaccounted for in energy attribution.

Tycho addresses these limitations through a refined kernel-level design. New tracepoints capture hard and soft interrupts, while extended per-CPU state tracking distinguishes between user processes, kernel threads, and idle execution. Each CPU maintains resettable bins that accumulate idle and interrupt durations within well-defined time windows, providing temporally bounded activity summaries aligned with energy sampling intervals. Cgroup identifiers are refreshed at every scheduling event to maintain accurate container attribution, even when processes migrate between control groups. The result is a stable, low-overhead data source that describes CPU usage continuously and with sufficient granularity to support fine-grained energy partitioning in the subsequent analysis.

1.16.2 Baseline and Architecture Overview

Kepler's kernel instrumentation consisted of a compact set of `eBPF` programs that sampled process-level CPU activity and a few hardware performance metrics. The core tracepoint, `tp_btf/sched_switch`, captured context switches and estimated per-process runtime by measuring the on-CPU duration between successive events. Complementary probes monitored page cache access and writeback operations, providing coarse indicators of I/O intensity. Hardware performance counters (CPU cycles, instructions, and cache misses) were collected through `perf_event_array` readers, enabling approximate performance characterization at the task level.

While effective for general profiling, this setup lacked the temporal resolution and system coverage required for precise energy correlation. The sampling process was driven entirely from userspace, leading to irregular collection intervals, and idle or interrupt time was never observed directly. Consequently, CPU utilization appeared complete only from a process perspective, leaving kernel and idle phases invisible to the measurement pipeline.

Tycho extends this architecture into a continuous kernel-side monitoring system. Each CPU maintains an independent state structure recording its current task, timestamp, and execution context. This allows uninterrupted accounting of CPU ownership, even between user-space scheduling events. New tracepoints for hard and soft interrupts measure service durations directly in the kernel, ensuring that all processor activity (user, kernel, or idle) is captured. Dedicated per-CPU bins accumulate these times within fixed analysis windows, which the userspace collector periodically reads and resets. Process-level metrics are stored in an LRU hash map, while hardware performance counters remain integrated via existing PMU readers.

In contrast to Kepler's snapshot-based sampling, Tycho's userspace collector consolidates all per-process and per-CPU deltas from the kernel maps once per polling interval into a single tick. This tick-based aggregation provides deterministic timing, reduces memory pressure, and guarantees temporal consistency across heterogeneous metric sources. Data therefore flows linearly from tracepoints to per-CPU maps and onward to the collector, forming a continuous and low-overhead measurement path that supports precise, time-aligned energy attribution.

1.16.3 Kernel Programs and Data Flow

Tycho's `eBPF` subsystem consists of a small set of tracepoints and helper maps that together maintain a continuous record of CPU activity. Each program updates per-CPU or per-task data structures in response to kernel events, ensuring that all processor time is accounted for across user, kernel, and idle contexts. The kernel side is event-driven and self-contained; aggregation into time-bounded ticks occurs later in userspace.

Scheduler Switch The central tracepoint, tp_btf/sched_switch, triggers whenever the scheduler replaces one task with another. It computes the elapsed on-CPU time of the outgoing process and updates its entry in the processes map, which stores cumulative runtime, hardware-counter deltas, and classification metadata such as cgroup_id, is_kthread, and command name. Hardware counters for instructions, cycles, and cache misses are read from preconfigured PMU readers at this moment, keeping utilization metrics temporally aligned with task execution. Each CPU also maintains a lightweight cpu_state structure that records the last timestamp, currently active PID, and task type. When the idle task (PID 0) is scheduled, this structure accumulates idle time locally, allowing continuous accounting even between user-space collection intervals. At polling time, the userspace collector drains these maps atomically, computing per-process deltas since the previous read and bundling all results into a single tick that represents the complete scheduler activity for that interval.

Interrupt Handlers To capture system activity outside user processes, Tycho introduces tracepoints for hard and soft interrupts. Pairs of entry and exit hooks (irq_handler_entry,exit and softirq_entry,exit) measure the time spent in each category by recording timestamps in the per-CPU state and adding the resulting deltas to dedicated counters. These durations are aggregated in cpu_bins, a resettable per-CPU array that also stores idle time. At each collection cycle, the userspace bpfCollector drains and resets these bins, incorporating their totals into the tick structure alongside the per-process deltas. This design maintains continuous coverage of kernel activity while preserving strict temporal alignment between CPU-state transitions and energy sampling.

Page-Cache Probes Kepler's original page-cache hooks (fexit/mark_page_accessed and tp/writeback_dirty_folio) are preserved. They increment per-process counters for cache hits and writeback operations, serving as indicators of I/O intensity rather than direct power consumption. These counters are read and reset as part of the same tick aggregation that handles scheduler and interrupt data.

Supporting Maps and Flow All high-frequency updates occur in per-CPU or LRU hash maps to avoid contention. pid_time_map tracks start timestamps for active threads, enabling precise runtime computation during context switches. processes holds per-task aggregates, while cpu_states and cpu_bins manage temporal accounting per core. PMU event readers for cycles, instructions, and cache misses remain shared with Kepler's implementation. At runtime, data flows from tracepoints to these maps and is drained periodically by the userspace collector, which consolidates the deltas into a single per-tick record before storing it in the ring buffer. This batched extraction forms a deterministic, lock-free telemetry path from kernel to analysis, ensuring high-frequency accuracy without per-event synchronization overhead.

1.16.4 Collected Metrics

The kernel eBPF subsystem exports a defined set of metrics describing CPU usage at process and system levels. These values are aggregated in kernel maps and periodically retrieved by the userspace collector for time-aligned energy analysis. Table 1.1 summarizes all metrics grouped by category.

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

All metrics are aggregated once per polling interval into a single userspace tick that contains per-process and per-CPU deltas. This tick-based representation replaces the former per-sample storage model, ensuring temporal consistency across metrics while retaining the semantics listed above.

Together these metrics form a coherent description of CPU activity. Time-based data quantify ownership of processing resources, hardware counters capture execution intensity, and classification attributes link activity to its origin. This dataset serves as the kernel-level foundation for energy attribution and higher-level modeling in userspace.

1.16.5 Integration with Energy Measurements

The data exported from the kernel define how CPU resources are distributed among processes, kernel threads, interrupts, and idle periods during each observation window. When combined with energy readings obtained over the same interval, these temporal shares provide the basis for proportional energy partitioning. Instead of relying on statistical inference or coarse utilization averages, Tycho attributes energy according to directly measured CPU ownership.

Each collection tick consolidates all per-process runtime and performance-counter deltas together with per-CPU idle and interrupt bins. The sum of these components represents the total active time observed by the processor during that tick, matching the energy sample boundaries defined by the timing engine. This strict temporal alignment ensures that every joule of measured energy can be traced to a specific class of activity—user workload, kernel service, or idle baseline. Through this mechanism, the `eBPF` subsystem provides the precise temporal structure required for fine-grained, container-level energy attribution in the subsequent analysis stages.

1.16.6 Efficiency and Robustness

The kernel instrumentation is designed to operate continuously with negligible system impact while ensuring correctness across kernel versions. All high-frequency data reside in per-CPU maps, eliminating cross-core contention and locking. Each processor updates only its local entries in `cpu_states` and `cpu_bins`, while per-task data are stored in a bounded LRU hash that automatically removes inactive entries. Arithmetic within tracepoints is deliberately minimal (timestamp subtraction and counter increments only) so that the added latency per event remains near the measurement noise floor.

Userspace retrieval employs batched `BatchLookupAndDelete` operations, reducing system-call overhead and maintaining constant latency regardless of map size. Hardware counters are accessed through pre-opened `perf_event_array` readers managed by the kernel, avoiding repeated setup costs. Each polling interval consolidates the collected deltas into a single userspace tick, ensuring deterministic timing and consistent aggregation across all CPUs. This architecture allows the subsystem to record thousands of context switches per second while keeping CPU overhead low.

Correctness is maintained through several safeguards. CO-RE (Compile Once, Run Everywhere) field resolution protects the program from kernel-version differences in `task_struct` layouts. Cgroup identifiers are refreshed only for the newly scheduled task, ensuring accurate container labeling even when group membership changes. The idle task (PID 0) and kernel threads are handled explicitly to prevent user-space misattribution, and the resettable bin design enforces strict temporal separation between collection ticks. Together, these measures yield a stable and version-tolerant tracing layer that can run indefinitely without producing inconsistent or overlapping tick data.

1.16.7 Limitations and Future Work

Although the extended `eBPF` subsystem provides comprehensive temporal coverage of CPU activity, several limitations remain. Its precision is ultimately bounded by the granularity of available energy telemetry, as energy readings must be averaged over fixed collection intervals to remain stable. Within shorter ticks, power fluctuations introduce noise that limits the accuracy of direct attribution.

The current implementation also omits processor C-state and frequency information. While idle and active time are distinguished, variations in power state and dynamic frequency scaling are not yet represented in the collected data. Including tracepoints such as `power:cpu_idle` and `power:cpu_frequency` would enable finer correlation between CPU state transitions and power usage. Additionally, very short-lived processes may terminate and be removed from the LRU map before the next tick is collected, leading to a slight underrepresentation of transient workloads.

1.17 GPU Collector Integration

1.17.1 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 (§ 1.21). 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*[12], 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 (§ 1.22.1).

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.

1.17.2 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 (§ 1.21, § 1.22.1).

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 lock-step 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 1.1 (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.

1.17.3 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.

1.17.4 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 $t_{\mathrm{obs},k}$ 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

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

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

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

The deviation

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

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

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

where n is the smallest non-negative integer such that $t_{\mathrm{next}} \geq t_{\mathrm{now}}$. 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}:

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

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 1.1 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.

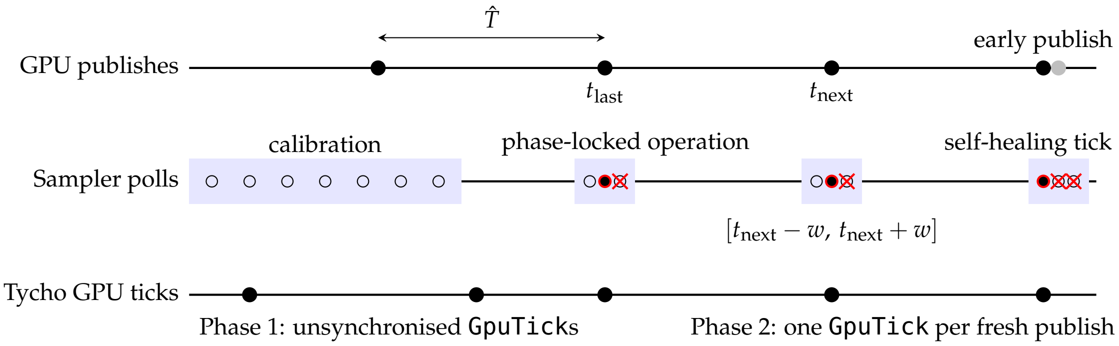

FIGURE 1.1: Phase-aware GPU polling timeline

1.17.5 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 (§ 1.17.3, § 1.17.4). 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 (§ 1.17.6). 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 (§ 1.22.1). 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.

1.17.6 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 (§ 1.21), 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 $t^{(i)}_{\text{last}} \leftarrow t^{(i)}_{\text{now}}$, 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.

1.17.7 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 (§ 1.17.6). 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 1.2 and 1.3 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 1.2: 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 1.3: Process-level metrics collected over a backend-defined time window.

1.17.8 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 (§ 1.21). 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.

1.17.9 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[13], 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.

1.18 RAPL Collector Integration

1.18.1 Scope and Baseline

RAPL-based telemetry provides CPU energy readings at the level of logical power domains such as package, cores, uncore, and DRAM. Tycho reuses Kepler's existing RAPL implementation almost unchanged: discovery, scaling, and raw counter reads are still provided by the shared `components` package (via the `power`-interface), which wraps the kernel's `powercap` interface and related mechanisms. This collector therefore does not introduce a new hardware access path but instead focuses on integrating the existing RAPL metrics into Tycho's timing and buffering model.

1.18.2 Architecture and Data Flow

The collector is deliberately simple and stateless. On each engine tick, the `Collect` function:

1. Verifies that system-wide RAPL collection is supported via `components.IsSystemCollectionSupported()`.
2. Obtains a snapshot of cumulative energy counters for all sockets and domains through `components.GetAbsEnergyFromNodeComponents()`.
3. Translates the per-socket results into a `RaplTick` structure containing:
   - a monotonic timestamp derived from `clock.Mono`, and
   - a map from socket identifier to domain counters.
4. Pushes exactly one immutable `RaplTick` into the shared ring buffer.

The tick stores raw, monotonically increasing counters in millijoules rather than precomputed deltas. Downstream analysis components compute inter-tick differences, perform wraparound handling, and align RAPL energy with CPU activity and platform-level power traces. This separation keeps the collector minimal while allowing the analysis layer to apply a consistent attribution strategy across all energy domains.

1.18.3 Collected Metrics

Table 1.4 summarises the metrics stored in each `RaplTick`.
All counters are subject to platform availability.

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

These metrics provide a compact, hardware-backed view of CPU and memory-subsystem energy at socket granularity. Combined with the timing guarantees of Tycho's engine and ring buffer, they form the CPU-related energy baseline against which per-process activity and platform-level power are later correlated.

1.18.4 Limitations and Reuse

Because RAPL domain availability is hardware dependent, not all fields in Table 1.4 are present on every platform. Some systems expose only package-level counters, while others additionally provide core, uncore, or DRAM domains. The collector reports only what the hardware exposes and does not attempt to approximate or reconstruct missing values.

Beyond this domain variability, no further limitations apply. The collector is intentionally minimal and serves as a timing-aware wrapper around the raw counters, providing reliable cumulative energy readings without adding intermediate modelling or transformation.

1.19 Redfish Collector Integration

The Redfish collector retrieves node-level power measurements from the server's Baseboard Management Controller (BMC) through the Redfish API. As an out-of-band source, it complements in-band telemetry such as RAPL by providing a hardware-validated view of total chassis power. Tycho integrates these measurements into its global timing framework, ensuring that BMC-sourced data can be correlated with all other collectors using a common monotonic clock.

1.19.1 Overview and Objectives

Redfish power readings are updated asynchronously and often with vendor-specific timing behaviour. Tycho therefore focuses on robustness, controlled polling, and precise timestamping rather than high-frequency sampling. The collector executes a single BMC query at the global engine cadence, extracts the current chassis power value, and emits a sample only when necessary. All readings are timestamped with Tycho's monotonic timebase and annotated with freshness metadata to express their temporal accuracy.

1.19.2 Baseline in Kepler

Kepler's original Redfish integration consisted of a periodic background ticker that queried the BMC at fixed intervals, typically every sixty seconds. It retrieved only instantaneous chassis power and made no distinction between new and repeated measurements. No mechanism existed to detect stale data, associate readings with BMC timestamps, or align Redfish values with other collectors. The Redfish implementation was therefore sufficient for coarse system-power reporting, but not suitable for the finer temporal structure required by Tycho.

1.19.3 Refactoring and Tycho Extensions

1.19.3.1 Timing Ownership and Polling Control

Tycho removes Kepler's internal Redfish ticker entirely. Polling is performed exclusively by Tycho's global timing engine, which invokes the collector once per engine tick. This ensures that all Redfish interactions occur in lockstep with the rest of the measurement pipeline and that every sample can be aligned unambiguously with simultaneous energy and utilization values from other collectors.

The polling frequency itself is never altered by the Redfish collector. Tycho maintains deterministic, externally selected polling intervals regardless of BMC behaviour.

1.19.3.2 Sequence-Based Newness Detection

Many BMCs return identical payloads for extended periods and only publish new power values intermittently. To avoid emitting redundant samples, Tycho uses a per-chassis sequence number provided by the Redfish client. Each measurement is considered fresh only when this sequence number differs from the one observed in the previous engine tick.

No header-based (for example `ETag`) or value-delta heuristics are used. Newness is driven solely by this explicit sequence value, ensuring a consistent and vendor-agnostic update mechanism.

1.19.3.3 Heartbeat Mechanism and Freshness Metric

Because BMC refresh cycles can be irregular, Tycho maintains a heartbeat window for each chassis. If no new Redfish sample has appeared within this window, the collector emits a heartbeat sample that carries forward the last known power value. This prevents gaps in the power time series while avoiding unnecessary fabrication of data.

Each emitted sample includes a freshness metric, defined as the time difference between the BMC-reported timestamp (when available) and the local collection time. This value quantifies the latency of the underlying Redfish pipeline. Freshness does not influence newness detection; it is an informational quality indicator for the analysis layer.

1.19.3.4 Fixed vs Auto Heartbeat Mode

Tycho supports two Redfish operating modes, selected by `TYCHO_REDFISH_POLL_AUTOTUNE`. Both modes perform BMC polling strictly at the global engine cadence.

In fixed mode, the heartbeat window is defined entirely by the user through `TYCHO_REDFISH_HEARTBEAT_MS`. This guarantees deterministic behaviour and is appropriate for reproducible benchmarks.

In auto mode, Tycho adapts the heartbeat window to the BMC's observed update pattern. Whenever a fresh sequence number is detected, the collector measures the inter-arrival duration to the previous update and stores it in a sliding window. The median of this window is taken as the representative publication period, and the heartbeat timeout is set to approximately 1.5 times this value, clamped between conservative bounds. This technique aligns Tycho's emission behaviour with the hardware's natural update rhythm while retaining deterministic polling.

1.19.4 Collected Metrics

The Redfish collector emits one record per chassis whenever a new measurement is detected or a heartbeat event occurs. Each record is timestamped using Tycho's monotonic clock and annotated with metadata relevant for temporal correlation. Table 1.5 summarises the collected fields.

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 1.5: Metrics collected by the Redfish collector.

Energy values such as `EnergyMilliJ` are computed downstream by integrating the power series and are not directly produced by the collector.

1.19.5 Integration and Data Flow

The Redfish collector is a passive component within Tycho's unified collection pipeline. At each engine tick it performs one BMC query, evaluates newness and heartbeat conditions, attaches monotonic timestamps, and writes the resulting record into a synchronized ring buffer. No additional processing occurs within the collector; all energy computations and correlations across metrics are performed in Tycho's analysis stage.

1.19.6 Accuracy and Robustness Improvements

Tycho improves Redfish robustness by combining explicit sequence-based newness tracking, adaptive heartbeat windows, and freshness estimation. Sequence tracking ensures that repeated BMC responses do not produce redundant samples. Adaptive heartbeat windows accommodate BMCs with slow or irregular update cycles. Freshness metadata exposes the latency of Redfish timestamps, allowing the analysis layer to assess the temporal reliability of each reading.

These measures provide stable and chronologically consistent power telemetry across heterogeneous BMC implementations.

1.19.7 Limitations

Redfish sampling remains limited by the update rate and timestamp quality of the underlying BMC. Most implementations refresh power data at intervals on the order of one to two seconds. Redfish does not report component-level power breakdowns; only total chassis power is available. For fine-grained attribution, Tycho relies on complementary in-band collectors such as RAPL and eBPF.

1.20 Configuration Management

1.20.1 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 § 1.21.

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.

1.20.2 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.

1.20.3 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.

1.20.3.1 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.

1.20.4 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.

1.20.5 Available Parameters

All parameters are read at startup and remain constant throughout execution. The following table 1.6 summarizes the user-facing configuration variables with their default values and functional scope. Internal or experimental parameters are omitted for clarity.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
CHECK THIS TABLE BEFORE HANDIN, THIS NEEDS CLEANUP
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

1.21 Timing Engine

1.21.1 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

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 1.6: User-facing configuration variables available in Tycho.

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 § 1.22, which retain recent histories for downstream integration and attribution.

1.21.2 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.

1.21.3 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.

1.21.4 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.

1.21.5 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, § 1.22, describes how these samples are buffered and retained for subsequent analysis, completing the temporal layer that underpins Tycho's measurement and attribution framework.

1.22 Ring Buffer Implementation

1.22.1 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.

1.22.2 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.

1.22.3 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.

1.22.4 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.

1.22.5 Polling frequency Calibration

1.22.5.1 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.

1.22.5.2 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[14, Table 2-2]. Jay et al. found that sampling slower than 50 Hz keeps relative error below 0.5%[15]. 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.

1.22.5.3 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.

1.22.5.4 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.

1.22.6 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.

1.22.6.1 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[16], 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[17], 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.

1.22.6.2 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[18], 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[19]. 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.

1.22.6.3 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[20], 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.

1.22.6.4 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.

1.23 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.

1.23.1 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.

1.23.2 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.

1.23.3 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.

1.23.4 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 § 1.23.5.

1.23.4.1 Collected Metrics

The following table 1.7 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 1.7: Process metadata collected by the process collector

1.23.5 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.

1.23.5.1 Collected Metrics

The kubelet collector records per-pod and per-container metadata as shown in Tables 1.8 and 1.9. 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 1.8: 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 1.9: Container metadata collected by the kubelet collector

1.23.6 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 [21].

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] International Energy Agency. *Energy and AI*. Licence: CC BY 4.0. Paris, 2025. URL: https://www.iea.org/reports/energy-and-ai.

[3] Ryan Smith. *Intel's CEO Says Moore's Law Is Slowing to a Three-Year Cadence — But It's Not Dead Yet*. Accessed: 2025-04-14. 2023. URL: https://www.tomshardware.com/tech-industry/semiconductors/intels-ceo-says-moores-law-is-slowing-to-a-three-year-cadence-but-its-not-dead-yet.

[4] Martin Keegan. *The End of Dennard Scaling*. Accessed: 2025-04-14. 2013. URL: https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/.

[5] Uptime Institute. *Global PUEs – Are They Going Anywhere?* Accessed: 2025-04-14. 2023. URL: https://journal.uptimeinstitute.com/global-pues-are-they-going-anywhere/.

[6] Eric Masanet et al. "Recalibrating global data center energy-use estimates". In: *Science* 367.6481 (2020), pp. 984–986. DOI: 10.1126/science.aba3758. eprint: https://www.science.org/doi/pdf/10.1126/science.aba3758. URL: https://www.science.org/doi/abs/10.1126/science.aba3758.

[7] Amit M. Potdar et al. "Performance Evaluation of Docker Container and Virtual Machine". In: *Procedia Computer Science* 171 (2020), pp. 1419–1428. ISSN: 1877-0509. DOI: 10.1016/j.procs.2020.04.152.

[8] Roberto Morabito. "Power Consumption of Virtualization Technologies: An Empirical Investigation". In: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. Dec. 2015, pp. 522–527. DOI: 10.1109/UCC.2015.93. (Visited on 05/21/2025).

[9] 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.

[10] Hubblo. *Scaphandre: Energy consumption monitoring agent*. https://github.com/hubblo-org/scaphandre. Accessed: 2025-06-24. 2025.

[11] 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.

[12] 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).

[13] 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/.

[14] 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.

[15] 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).

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

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

[18] 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).

[19] 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).

[20] 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).

[21] 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).