Zurich University of Applied Sciences
Department School of Engineering
Institute of Computer Science

MASTER THESIS

Title

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:	Title
Author:	Caspar Wackerle
Date:	January 31, 2026
Keywords:	energy efficiency, 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

Chapter 1

Introduction and Context

1.1 System Environment for Development, Build and Debugging

This section documents the environment used to develop, build, and debug *Tycho*; detailed guides live in [2].

1.1.1 Host Environment and Assumptions

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

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

1.1.2 Build Toolchain

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

1.1.2.1 Local builds

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

1.1.2.2 Container images

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

1.1.2.3 Continuous integration

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

1.1.2.4 Versioning and reproducibility

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

1.1.3 Debugging Environment

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

1.1.3.1 Remote debugging setup

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

1.1.3.2 Integration with the editor

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

The editor attaches over the SSH-forwarded DAP port; the inner loop is build locally with `make`, launch under Delve with a DAP listener, attach via SSH, inspect, adjust, repeat. When the goal is to validate behavior in a cluster setting rather than to step through code, the deploy oriented path is used instead. In that case the image is built and pushed, and observation relies on logs and metrics rather than an attached debugger.

1.1.3.3 Limitations and challenges

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

1.1.4 Supporting Tools and Utilities

1.1.4.1 Configuration and local orchestration

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

1.1.4.2 Container, cluster, and monitoring utilities

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

1.1.5 Relevance and Limitations

1.1.5.1 Scope and contribution

The development, build, and debugging environment described in § 1.1.2 and § 1.1.3 is enabling infrastructure rather than a scientific contribution. Its purpose is to make modifications to *Tycho* feasible and to support evaluation, not to advance methodology in software engineering or tooling.

Documenting the environment serves reproducibility and auditability. A reader can verify that results were obtained on bare-metal with access to the required telemetry, and can reconstruct the build pipeline from source to binary and container image. The references to the repository at the start of this section in § 1.1 provide the operational detail that is intentionally omitted from the main text.

1.1.5.2 Boundaries and omissions

Installation steps, editor-specific configuration, system administration, security hardening, and multi tenant policy are out of scope; concrete commands live in the repository. Where concrete commands matter for reproducibility they are available in the repository documentation cited in § 1.1.

1.2 ebpf-collector-Based CPU Time Attribution

1.2.1 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.

1.2.2 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.

Data flows linearly from tracepoints to per-CPU maps and onward to the userspace collector, forming a continuous and low-overhead measurement path. This architecture transforms Kepler's periodic snapshot model into a streaming telemetry layer that maintains temporal consistency and provides the necessary basis for accurate, time-aligned energy attribution.

1.2.3 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.

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 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 sampling intervals.

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, userspace reads these bins, derives total CPU activity for the window, and resets them to zero for the next interval.

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.

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 then to the userspace collector through batched lookups, forming a deterministic, lock-free telemetry path from kernel to analysis.

1.2.4 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 CPU. Time with no runnable task (PID 0).
IRQ time	`irq_handler_{entry,exit}`	Per CPU. Duration spent in hardware interrupt handlers.
SoftIRQ time	`softirq_{entry,exit}`	Per CPU. 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 CPU. Frequency of specific soft interrupt vectors.

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

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.

1.2.5 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 process contributes its accumulated runtime and performance-counter deltas, while system activity and idle phases are derived from the per-CPU bins. The sum of these components represents the total active time observed by the processor, matching the energy sample boundaries defined by the timing engine. This 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.

1.2.6 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. 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 sampling windows. Together, these measures yield a stable and version-tolerant tracing layer that can run indefinitely without producing inconsistent or overlapping samples.

1.2.7 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 sampling windows to remain stable. Within shorter intervals, 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 be evicted from the LRU map before collection, slightly undercounting transient workloads.

1.3 GPU Collector Integration

The GPU collector extends Tycho's measurement framework to include accelerator energy and utilization data. Building on Kepler's existing device abstraction, it reuses and refines NVIDIA-specific collection paths while integrating them into Tycho's modular timing and buffering architecture.

1.3.1 Overview and Objectives

The GPU collector extends Tycho's energy measurement framework to include accelerator telemetry. Its purpose is not to introduce new metrics, but to integrate existing GPU energy and utilization data into Tycho's synchronized collection cycle. By reusing and refining Kepler's accelerator interface, Tycho can obtain GPU-level power and activity data without duplicating existing logic.

The collector operates identically across hardware tiers. On systems equipped with NVIDIA's DCGM or NVML interfaces, it retrieves instantaneous power, utilization, and memory metrics, aligning them with Tycho's monotonic timebase for unified analysis with CPU and platform energy data.

1.3.2 Architecture and Backend Selection

Kepler's accelerator abstraction exposes a uniform device interface backed by multiple telemetry providers. Tycho reuses this structure to maintain compatibility and minimize maintenance effort. Two NVIDIA backends are supported: `DCGM` (Data Center GPU Manager) and `NVML` (NVIDIA Management Library).

DCGM is preferred when available, as it provides high-resolution telemetry, process-level utilization, and Multi-Instance GPU (MIG) awareness on enterprise hardware. NVML serves as a fallback for consumer-grade devices with limited instrumentation. This layered design ensures that Tycho can operate across development and production environments without configuration changes.

Through this abstraction, the GPU collector accesses metrics through a single interface. Backend selection, device enumeration, and capability handling are managed internally, allowing Tycho to treat GPU data as a consistent input source, regardless of the underlying driver.

1.3.3 Collected Metrics

The GPU collector retrieves instantaneous and cumulative telemetry from the active accelerator backend (`DCGM` or `NVML`). All values are sampled at fixed intervals and aligned to Tycho's monotonic timebase. Table 1.2 lists the available input metrics.

Metric	Unit	Description
Utilization metrics		
`SMUtilPct`	%	Percentage of active streaming multiprocessors (SMs), representing compute load.
`MemUtilPct`	%	GPU memory controller utilization.
`EncUtilPct`	%	Hardware video encoder utilization.
`DecUtilPct`	%	Hardware video decoder utilization.
Energy and thermal metrics		
`PowerMilliW`	mW	Instantaneous power draw per device.
`EnergyMicroJ`	µJ	Integrated energy derived from power samples over time.
`TempC`	°C	Current GPU temperature.
Memory and frequency metrics		
`MemUsedBytes`	bytes	Allocated frame-buffer memory.
`MemTotalBytes`	bytes	Total available frame-buffer memory.
`SMClockMHz`	MHz	Streaming multiprocessor clock frequency.
`MemClockMHz`	MHz	Memory clock frequency.

TABLE 1.2: Metrics collected by the GPU collector.

Together these metrics describe the GPU's operational state and power consumption. They provide the foundation for process-level energy attribution by combining instantaneous power with per-process utilization data retrieved from the same backend.

1.3.4 Integration and Data Flow

The GPU collector operates as an independent module within Tycho's collection framework. Initialization occurs during startup, where Kepler's accelerator registry detects available devices and activates either the DCGM or NVML backend. Once initialized, the collector periodically polls device metrics using Tycho's scheduling engine and stores each result in a synchronized ring buffer.

Each collected sample is timestamped through the system's monotonic clock to maintain temporal consistency with other subsystems such as RAPL and eBPF. This design allows GPU data to be directly correlated with CPU and platform measurements during post-processing. By aligning all sources under a shared timebase and buffer structure, Tycho guarantees consistent sampling intervals and deterministic integration across heterogeneous energy domains.

1.3.5 Robustness and Limitations

The collector was designed for stability across varying hardware and driver configurations. Additional validation and error handling were introduced to tolerate missing or partially initialized devices, ensuring safe operation even when GPUs are unavailable or the driver interface is incomplete. The NVML backend was slightly refactored for safer initialization and shutdown semantics. Device enumeration and map handling were hardened to prevent stale handles or nil dereferences during partial driver availability, improving resilience when GPUs are not yet ready or temporarily absent. These changes primarily improve reliability rather than extend measurement scope.

Process-level resolution depends on backend capabilities. Enterprise GPUs exposed through DCGM support per-process utilization, while consumer-grade devices using NVML typically provide aggregate values only. This limitation affects attribution accuracy but does not compromise energy sampling itself.

The current implementation was validated on a consumer GPU; full verification on data-center hardware will follow once suitable test systems become available. Despite these differences, the collector provides stable and temporally precise GPU telemetry, completing Tycho's set of primary energy input sources.

1.4 Redfish Collector Integration

The Redfish collector retrieves node-level power data from the server's Baseboard Management Controller (BMC) via the Redfish API. As an out-of-band source, it complements in-band interfaces such as RAPL by providing an external, hardware-validated view of total system power. Tycho integrates this telemetry into its synchronized measurement framework, ensuring consistent timing and comparability across collectors.

1.4.1 Overview and Objectives

Redfish power metrics are vendor-defined and updated asynchronously, with variable latency and precision. The Tycho implementation therefore focuses on reliability, timing control, and consistent timestamping. All polling, freshness tracking, and temporal alignment are managed centrally by Tycho, allowing Redfish samples to be merged with other data sources for later workload-level energy attribution.

1.4.2 Baseline in Kepler

In Kepler, the Redfish implementation provided a minimal wrapper around the BMC's `/redfish/v1/Chassis/*/Power` endpoint. Its sole purpose was to retrieve aggregated chassis power at a fixed interval and expose it through the node-level energy interface used by the power model. The default polling frequency was set to 60 seconds, adequate for coarse monitoring but too infrequent for detailed analysis.

At such long intervals, issues like repeated values or timing drift were largely masked by the coarse sampling period. However, the design offered no mechanisms to detect new versus stale data, to associate samples with BMC timestamps, or to align readings precisely with other metrics. The internal background ticker operated independently of other Kepler collectors, providing no unified notion of time or freshness. Kepler's Redfish integration was therefore sufficient for low-resolution system energy reporting, but not designed for higher measurement intervals or fine-grained temporal correlation.

1.4.3 Refactoring and Tycho Extensions

1.4.3.1 Timing Ownership and Polling Control

Tycho removes Kepler's internal ticker and delegates all Redfish polling to its centralized timing engine. To account for the unpredictable nature of BMC update cycles, Tycho introduces an optional adaptive mode governed by `TYCHO_REDFISH_POLL_AUTOTUNE`. When enabled, the collector dynamically infers a suitable polling interval from observed publication gaps, learning the effective refresh frequency of the specific Redfish implementation. When disabled, Tycho performs fixed-interval polling strictly at the user-defined cadence, preserving deterministic operation.

By externalizing timing control, the collector decouples sampling from Redfish's internal pacing, enabling reproducible experiments and consistent temporal correlation with other measurement sources.

1.4.3.2 Header-Based Newness and Sequence Tracking

When polled at higher frequencies, Redfish endpoints often repeat identical payloads until the BMC updates its internal sensors. To avoid redundant samples, Tycho introduces a lightweight newness detection mechanism combining HTTP headers and value comparison.

Each response is inspected for the `ETag` and `Date` headers. If an `ETag` differs from the previously stored value, or if the `Date` timestamp is newer, the sample is treated as fresh. If no header change is observed, Tycho falls back to value-based detection by comparing the reported power against the previous reading. A monotonically increasing `seq` counter is maintained per chassis to mark every distinct update, allowing downstream components to identify repeated or skipped readings unambiguously.

This design provides consistent differentiation between new and stale measurements without requiring vendor-specific heuristics. It also ensures that timestamp alignment and freshness analysis remain reliable even when Redfish responses arrive irregularly or contain repeated values.

1.4.3.3 Heartbeat Mechanism and Freshness Metric

Because Redfish publication intervals can vary considerably between BMC implementations, Tycho introduces a heartbeat mechanism to ensure continuous sample availability. If no new data are received within a configurable timeout, defined by `TYCHO_REDFISH_HEARTBEAT_MAX_GAP_MS`, the collector emits a heartbeat sample that reuses the last known power value. This prevents temporal gaps in the time series and maintains a consistent data flow for later energy integration.

Each emitted sample also carries a *freshness* metric, representing the time difference between the Redfish-reported `Date` header (if present) and the local collection timestamp. This value quantifies the staleness of a reading and allows the analysis layer to account for delayed or buffered updates. In practice, freshness remains below one second on well-behaved BMCs but can increase significantly under heavy load or poor firmware timing.

Together, the heartbeat and freshness metric allow Tycho to stabilize asynchronous Redfish data streams and provide temporal confidence estimates for each sample.

1.4.3.4 Fixed vs Auto Polling Mode

Tycho supports two complementary Redfish polling strategies, selectable via `TYCHO_REDFISH_POLL_AUTOTUNE`. In *fixed mode* (`false`), polling occurs strictly at the user-defined cadence `TYCHO_REDFISH_POLL_MS`. This mode guarantees deterministic timing and is suited for controlled experiments where Redfish irregularities are tolerable or where timing synchronization with other collectors is critical.

When *auto mode* (`true`) is enabled, the collector dynamically adjusts its internal expectations to match the observed publication rhythm of the BMC. It derives the median inter-arrival time of new Redfish samples and adapts the expected heartbeat gap accordingly. This allows the collector to align its emission behavior with the actual update frequency of the hardware, minimizing redundant polls and improving temporal coherence between samples.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

THIS SECTION NEEDS TO BE UPDATED UPON CALIBRATION PACKAGE COMPLETION
A future calibration module will further refine `POLL_MS` and related delay parameters based on startup profiling, but this mechanism remains outside the collector itself. Within Tycho, the auto mode provides a self-stabilizing behavior that balances responsiveness with measurement overhead, while the fixed mode ensures reproducibility for benchmark-oriented studies. XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

1.4.4 Collected Metrics

The Redfish collector retrieves instantaneous power readings from the BMC for each physical chassis. Unlike software-based collectors, it provides direct hardware telemetry at node level and does not expose component-level breakdowns. All readings are timestamped, aligned to Tycho's global monotonic clock, and supplemented with freshness and sequence metadata to support temporal correlation. Table 1.3 lists the collected fields.

These metrics provide a coherent node-level view of power consumption with explicit temporal context, forming the hardware baseline for higher-level attribution in later stages of the Tycho pipeline.

1.4.5 Integration and Data Flow

The Redfish collector operates as a passive data source within Tycho's unified collection framework. It queries the BMC through the Redfish API, extracts instantaneous

Metric	Unit	Description
Primary power metrics		
`PowerWatts`	W	Instantaneous chassis power draw reported by the BMC via `/redfish/v1/Chassis/*/Power`.
`EnergyMilliJ`	mJ	Integrated node energy derived from consecutive power samples (computed downstream).
Temporal and identity metadata		
`ChassisID`	-	Identifier of the chassis or enclosure corresponding to the Redfish endpoint.
`Seq`	-	Incremental counter marking each new reading as determined by header or value changes.
`SourceTime`	s	Original BMC timestamp parsed from the HTTP `Date` header, if available.
`CollectorTime`	s	Local collection time according to Tycho's monotonic clock.
`FreshnessMs`	ms	Time difference between `SourceTime` and `CollectorTime`, indicating sample latency.
Operational context		
`Heartbeat`	flag	Marks a repeated emission when no new BMC data were available within the configured heartbeat interval.
`PollMode`	enum	Indicates whether fixed or auto polling mode was active during sampling.

TABLE 1.3: Metrics collected by the Redfish collector.

chassis power, and writes each result into a synchronized ring buffer shared with the central engine. Each record carries both system- and collection-time metadata, enabling later temporal alignment during analysis.

This integration layer is deliberately lightweight: the collector's responsibility ends once valid samples are obtained and buffered. All subsequent processing is handled by Tycho's analysis modules. This separation keeps the collector simple, minimizes coupling to higher layers, and isolates potential BMC irregularities from the rest of the system.

1.4.6 Accuracy and Robustness Improvements

Tycho introduces several measures to improve the precision and reliability of Redfish telemetry compared to Kepler. Header-based newness detection ensures that only genuinely updated readings are processed, reducing redundant samples caused by repeated BMC responses. Each reading carries a freshness metric that quantifies its temporal distance from the BMC's internal timestamp, providing explicit visibility into data latency. A lightweight heartbeat mechanism compensates for occasional gaps or stalls in BMC reporting, maintaining continuity in the power time series without fabricating new information.

These measures collectively enhance stability across heterogeneous Redfish implementations and ensure that all retained samples are both valid and chronologically consistent.

1.4.7 Limitations

Despite its improved design, the Redfish collector remains constrained by the capabilities and responsiveness of the underlying BMC. Sampling frequency is typically limited to one or two seconds, and the precision of reported timestamps varies widely across vendors. No component-level breakdown is available (only total chassis power), restricting fine-grained attribution to software-based collectors such as RAPL or eBPF.

1.5 Configuration Management

1.5.1 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 § **??**.

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.

1.5.2 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.

1.5.3 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.

1.5.3.1 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.

1.5.4 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.

1.5.5 Available Parameters

All parameters are read at startup and remain constant throughout execution. The following table 1.4 summarizes the user-facing configuration variables with their default values and functional scope. Internal or experimental parameters are omitted for clarity.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
CHECK THIS TABLE BEFORE HANDIN, THIS NEEDS CLEANUP
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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 1.4: User-facing configuration variables available in Tycho.

1.6 Timing Engine

1.6.1 Overview and Motivation

Tycho introduces a dedicated timing engine that replaces the synchronous update loop used in Kepler with an event-driven, per-metric scheduling layer. While the conceptual motivation for this change was discussed in § **??**, its practical purpose is straightforward: to decouple the collection frequencies of heterogeneous telemetry sources and to establish a common temporal reference for subsequent analysis.

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

The timing engine thus provides the temporal backbone of Tycho: it defines *when* each collector produces samples and ensures that all samples can later be correlated on a unified, monotonic timeline. Collected samples are pushed immediately into per-metric ring buffers, described in § 1.7, which retain recent histories for downstream integration and attribution.

1.6.2 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 epcoh, 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.

1.6.3 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.

1.6.4 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.

1.6.5 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, § 1.7, describes how these samples are buffered and retained for subsequent analysis, completing the temporal layer that underpins Tycho's measurement and attribution framework.

1.7 Ring Buffer Implementation

1.7.1 Overview

Tycho employs a per-metric ring buffer to store recent samples 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 samples 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.

1.7.2 Data Model and Sample Types

Each ring buffer is strongly typed and holds a single metric-specific sample structure. These sample types encapsulate the data collected by their respective sources and all embed the `SampleMeta` structure, which records Tycho's monotonic epoch. Depending on the metric, samples may contain simple scalar values or more complex objects such as maps or nested counters. For example, a `RaplSample` stores per-socket energy readings in a map of domains, while `BpfSample` includes process-level counters and hardware event deltas. This typed approach simplifies access and ensures that all samples, regardless of complexity, can be correlated on a uniform temporal axis defined by the timing engine.

1.7.3 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 samples required to represent the desired time window and adds a small margin of spare capacity to tolerate irregular sampling or short bursts of delayed ticks. 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.

1.7.4 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 sample insertion. In practice, collectors append samples 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.

Appendix A

Appendix Title

Bibliography

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

[2] 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`.