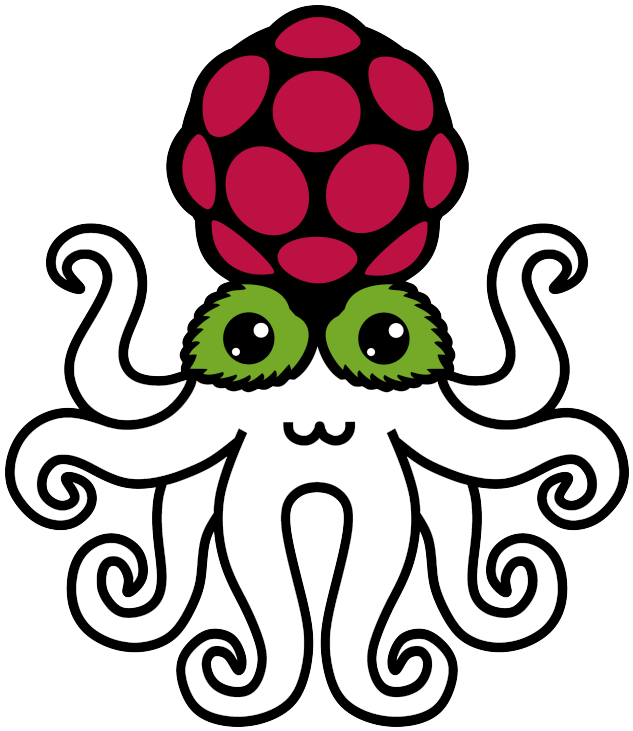
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Beyond the Data Center: Distributed Computing on a Raspberry Pi 5 Cluster

Bachelor's thesis in Computer Science and Engineering

Livia Borg

Emil Burman

Axel Forsberg

Mathias Fredriksson

Emily Tiberg

Filip Westman

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Bachelor's thesis 2026

Beyond the Data Center: Distributed Computing on a Raspberry Pi 5 Cluster

Livia Borg | Emil Burman | Axel Forsberg

Mathias Fredriksson | Emily Tiberg | Filip Westman

Department of Computer Science and Engineering
Division of Computer Science and Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Beyond the Data Center: Distributed Computing on a Raspberry Pi 5 Cluster

© Livia Borg, 2026. © Emil Burman, 2026. © Axel Forsberg, 2026.

© Mathias Fredriksson, 2026. © Emily Tiberg, 2026. © Filip Westman, 2026.

Supervisor: Fareed Mohammad Qararyah
Department of Computer Science and Engineering, Computer and Network Systems

Examiner: Arne Linde
Department of Computer Science and Engineering, Computer and Network Systems

Graded by teacher: Roger Johansson
Department of Computer Science and Engineering, Computer and Network Systems

Bachelor's Thesis 2026
Department of Computer Science and Engineering
Division of Computer Science and Software Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Logotype for the project, created by Livia Borg

Typeset in LaTeX
Gothenburg, Sweden 2026

Abstract

Distributed computing clusters are commonly used to provide scalable computation and large memory capacity for demanding workloads. In recent years, single-board computers have become increasingly capable and power-efficient, making them an attractive low-cost alternative for building small-scale distributed systems. However, creating such clusters in a way that is scalable, practical, and user-friendly remains challenging due to limited hardware resources and the need for lightweight management and monitoring solutions.

This thesis investigates how a distributed computing cluster built from single-board computers can be made practical through lightweight orchestration and purpose-built observability tooling. A central contribution is a custom telemetry system designed for resource-constrained nodes, where existing monitoring solutions impose unnecessary I/O on storage-limited hardware and offer limited control over which metrics are collected and how frequently they are reported. The system collects, transmits and visualizes hardware and performance metrics in real time through a custom web based interface while imposing no measurable impact on workload performance.

To evaluate the system, a Raspberry Pi 5 cluster was constructed using Kubernetes for orchestration. Three workloads were deployed to stress different dimensions of the cluster: matrix multiplication for parallel compute throughput, distributed password recovery for CPU-intensive data parallelism, and split large language model inference for distributed memory capacity. The results show that the cluster achieved significant performance improvements compared to single-node execution, particularly for highly parallelizable workloads. The system also demonstrated good power efficiency and highlighted the advantages of distributed memory for running larger LLMs. However, the limited computational performance of individual Raspberry Pi nodes means that many devices are required to approach the performance of a conventional high-performance machine. Overall, the work demonstrates that single-board computer clusters can provide a flexible and energy-efficient platform for distributed computing, especially when combined with lightweight orchestration and observability tools.

Keywords: distributed computing, Raspberry Pi, cluster, Kubernetes, scalability, telemetry, observability, LLM inference, SBC.

Sammandrag

Distribuerade kluster används ofta för att tillhandahålla skalbar beräkningskapacitet och stort minnesutrymme för krävande arbetslaster. Under senare år har enkortsdatorer blivit allt mer kraftfulla och energieffektiva, vilket gör dem till ett attraktivt lågkostnadsalternativ för att bygga småskaliga distribuerade system. Att skapa kluster på ett sätt som är skalbart, praktiskt, och användarvänligt är dock fortfarande utmanande på grund av begränsade hårdvaruresurser och behovet av resurssnåla lösningar för klusterhantering och systemövervakning.

Detta arbete undersöker hur ett distribuerat kluster byggt av enkortsdatorer kan göras praktiskt genom effektiv orkestrering och specialutvecklade verktyg för observabilitet. Ett centralt bidrag är ett anpassat telemetrisystem utformat för resursbegränsade noder, där befintliga övervakningslösningar orsakar onödig I/O-belastning på lagringsbegränsad hårdvara. Utöver det erbjuds begränsad kontroll över vilka mätvärden som samlas in, och hur ofta de rapporteras. Detta system samlar in, överför och visualiserar hårdvaru- och prestandamätvärden i realtid genom ett anpassat webbaserat gränssnitt, utan att orsaka någon mätbar påverkan på arbetslasternas prestanda.

För att utvärdera systemet konstruerades ett Raspberry Pi 5-kluster med Kubernetes som orkestreringsplattform. Tre arbetslaster användes för att belasta olika delar av klustret: matrismultiplikation för parallell beräkningskapacitet, distribuerad lösenordsåterställning för CPU-intensiv dataparallellism, samt delad inferens av stora språkmodeller för att utnyttja ökad minneskapacitet. Resultaten visar att klustret uppnådde betydande prestandaförbättringar jämfört med exekvering på en enskild nod, särskilt för arbetslaster som kan parallelliseras effektivt. Systemet uppvisade även god energieffektivitet och demonstrerade fördelarna med distribuerat minne för att hantera större språkmodeller. Samtidigt innebär den begränsade beräkningskapaciteten hos enskilda Raspberry Pi-noder att många enheter krävs för att närma sig prestandan hos en konventionell högpresterande dator. Sammantaget visar arbetet att kluster av enkortsdatorer kan utgöra en flexibel och energieffektiv plattform för distribuerad beräkning, särskilt i kombination med effektiva verktyg för orkestrering och observabilitet.

Nyckelord: distribuerad beräkning, Raspberry Pi, kluster, Kubernetes, skalbarhet, telemetri, observabilitet, språkmodell, enkortsdator.

Acknowledgements

We would like to express our gratitude to our supervisor, Fareed Mohammad Qararyah, for helping and supporting us throughout our project. We would also like to extend our thanks to Pedro Petersen Moura Trancoso for helping us with ideas along the way, and for creating this project to begin with.

Contents

Glossary

Below is the list of technical terms and named technologies referenced throughout this thesis listed in alphabetical order.

Ansible	An open source automation tool that configures systems and deploys software by executing declarative instructions, written in YAML, against remote hosts over SSH.
Container	A lightweight, isolated execution environment that packages an application together with its dependencies and runs on a shared host operating system kernel.
CPU governor	A kernel component that controls how a processor scales its clock frequency in response to load, implementing a chosen policy such as maximizing performance or conserving power.
Cluster	A group of independent computers connected over a network and coordinated so that they can be used as a single computing resource.
CSS	Cascading Style Sheets; a language for describing the visual presentation of documents written in HTML, separating styling rules such as colors, fonts, and layout from the underlying page structure.
DaemonSet	A Kubernetes workload type that ensures a copy of a given container runs on every selected node in the cluster.
Docker	A platform for packaging an application together with its dependencies into a standardized container that runs consistently on any supported host.
Endpoint	A specific addressable location exposed by a network service, typically identified by a URL, at which a client can send requests.
Go	A statically typed, compiled programming language developed by Google, designed for concurrent network services and known for its small runtime and fast compilation.

Heterogeneous	Refers to a cluster with different hardware configurations for nodes. This can be in the form of architectures, or as specialized purpose-built hardware working together in a cluster with ordinary nodes.
Job	A Kubernetes workload type that runs one or more pods to completion, tracking their success and handling retries, used for finite tasks rather than long-running services.
Kubernetes	An open source system for automating the deployment, scaling, and management of containerized applications across a cluster of machines.
K3s	A lightweight distribution of Kubernetes designed for resource-constrained environments, packaged as a single binary and targeted at edge and embedded deployments.
Monorepo	A software development strategy in which the source code for multiple projects or components is stored together in a single version control repository.
Pod	The smallest deployable unit in Kubernetes, consisting of one or more containers that share storage, network, and a specification for how they should run.
Protocol Buffers	A language-neutral, binary serialization format developed by Google, in which message structures are defined in schema files from which source code is generated for multiple programming languages.
Schema	A formal description of the structure of data, specifying which fields exist, their types, and how they relate to one another.
Svelte	A JavaScript user interface framework that compiles components to standalone JavaScript at build time, avoiding the need for a runtime framework or a virtual DOM in the browser.
TailwindCSS	A CSS framework that simplifies design work and strips away unused CSS elements in production thereby optimizing storage usage.
WebSocket	A network protocol that provides a persistent, bidirectional communication channel between a client and a server over a single TCP connection, used for real-time data delivery.

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order.

API	Application programming interface
ARM	Advanced RISC machines
DOM	Document object model
gRPC	Google remote procedure call
HTTP	Hypertext transfer protocol
I/O	Input/Output
LAN	Local area network
LLM	Large language model
NFS	Network file system
NVMe	Non-volatile memory express
RPC	Remote procedure call
SBC	Single-board computer
SD	Secure digital
SSD	Solid-state drive

List of Figures

List of Tables

1. Introduction

Distributed computing is a foundational part of a modern digital society, allowing scalability of operations to a degree that has previously been impossible on single systems. The core idea of distributed computing is to enable multiple nodes (computers) to work together as a cohesive system to solve complex problems.

Many aspects of today's digital world, from large-scale data processing and cloud computing, to scientific simulations and real-time analytics, are driven by massive computer clusters. While these workloads are generally handled by large data centers, delegating tasks to them comes with a specific set of challenges including energy demand, latency overhead and privacy concerns for the users. As these concerns grow, so does interest in decentralized clusters.

Edge (at the source) and fog (at the local network) computing are meant to solve these issues by moving computing closer to the end user and reducing reliance on large data centers. Simultaneously, advancements in single-board computers (SBCs) have enabled the development of smaller, more power-efficient hardware capable of increasingly greater computational performance.

The goal of this project is to develop a local distributed system characterized by low latency, increased privacy and control over data, and reduced energy consumption. The system is also intended to be user-friendly while supporting scalability and the integration of additional hardware. These properties are evaluated against the advantages typically associated with large-scale data centers, namely performance and availability. The resulting analysis aims to determine how far such a system can be pushed in terms of performance, scalability, and energy efficiency.

1.1 Purpose

The purpose of the project is to investigate the practical viability of a local distributed system, as outlined above, using a cluster of Raspberry Pi 5 units as the foundation. Workloads are distributed across multiple low-power devices to evaluate system efficiency and energy-consumption across a variety of computing tasks. These tasks include matrix multiplication, password recovery using hashcat, and inference for large language models (LLMs). Moreover, the project aims to develop a lightweight custom monitoring system capable of providing real-time insight into the status of the cluster. A web-based dashboard serves as the user-facing interface, visualizing collected telemetry and making the system accessible without requiring direct interaction with the underlying infrastructure.

1.2 Research questions

This project investigates whether a local distributed computing cluster built from single-board computers can provide practical advantages such as privacy, flexibility, and reduced dependence on external infrastructure, while maintaining acceptable performance and energy efficiency. To evaluate this, different cluster configurations and optimization strategies are examined using the applications mentioned in Section 1.1, whose background is detailed in Section 2.5. These workloads represent varying computational and resource requirements.

The project is guided by a set of research questions related to energy efficiency, scalability, inter-node communication, and observability. These questions are explored through the design, implementation, and evaluation of the system.

1.2.1 Energy efficiency

A central question of this work is how energy efficient an SBC-based distributed cluster can be under both active and idle conditions. Energy efficiency depends not only on workload execution, but also on how effectively the system manages low-utilization states. While power consumption can be influenced through CPU configuration and software optimization, estimating realistic long-term efficiency remains challenging, particularly for clusters intended for intermittent or single-user workloads where utilization patterns vary over time.

1.2.2 Scalability

Another research question concerns how the computational throughput of the cluster scales as additional nodes are introduced. A practical distributed system must be able to integrate new nodes with minimal configuration overhead while continuing to distribute workloads efficiently. This requires orchestration mechanisms capable of dynamically detecting nodes, managing workload placement, and avoiding bottlenecks caused by centralized coordination or storage access. The project therefore investigates both the scaling characteristics of distributed workloads and the practicality of expanding the cluster architecture.

1.2.3 Inter-node communication

The project also examines whether network communication and storage throughput become limiting factors in a local SBC cluster. Distributed workloads depend on efficient communication between nodes, and insufficient bandwidth or excessive latency may reduce the benefits of parallel execution. In addition, the relatively limited storage performance of SD-card-based systems introduces the risk of I/O bottlenecks during file transfers and workload synchronization. Evaluating the relative impact of CPU limitations, networking, and storage throughput is therefore an important part of the study.

1.2.4 Telemetry

An additional research question is whether a custom telemetry and observability system can monitor the cluster in real time without measurably affecting workload performance. Existing monitoring solutions often generate unnecessary disk writes and resource overhead that may be unsuitable for storage-constrained SBC hardware. This project therefore investigates whether a lightweight, purpose-built observability pipeline can provide detailed real-time monitoring while maintaining minimal impact on the cluster itself.

1.3 Limitations

In order to ensure that the project remains feasible within the given time frame and resource constraints, certain limitations have been imposed. Redundancy for the master node and the storage is not explored due to hardware limitations. Network security is not considered, as the cluster operates on a local network. Finally, the operating system is limited to Raspberry Pi OS Lite, a lightweight distribution optimized for Raspberry Pi hardware, to ensure maximum resource availability.

1.4 Societal and ethical aspects

The development of localized, small-scale compute clusters introduces several ethical and societal dimensions. By enabling local execution of workloads, the system reduces the necessity of transmitting potentially sensitive data to third parties, aligning with *privacy-by-design* principles and ensuring that data remains under physical and digital control of the user. However, this implies a shift in responsibility, as users must maintain network security and system integrity without the support of a professional data center operator.

From a sustainability perspective, the use of single-board computers such as the Pi provides an opportunity to reduce the energy consumption associated with modern computing workloads. Finally, by utilizing affordable and commercially available components, this project contributes to the democratization of distributed computing. Lowering the economic barriers to entry may enable broader adoption in educational settings and small-scale research environments.

2. Theory

This chapter introduces the background concepts needed to understand the design and evaluation of the cluster, covering distributed computing, followed by single-board computers, node provisioning, shared storage, and cluster orchestration. It then describes the three evaluation workloads and their parallelism strategies, and closes with observability concepts relevant to the telemetry system.

2.1 Distributed computing

A distributed system is a collection of independent computers that cooperate over a network to act as a single computational resource. Distribution is the standard response when a single machine lacks the speed or memory for a given workload, since the aggregate resources of a cluster can exceed what any individual node could provide [1]. The same arrangement can improve reliability by replicating work or state so that the failure of any one node does not stop the system as a whole [2]. These benefits come at a cost: data must traverse a network orders of magnitude slower than local memory, work must be partitioned to keep nodes busy without contending for shared resources, and failures of nodes, links, or the coordinating layer must be detected and handled rather than allowed to corrupt the computation. Many of the engineering decisions made in distributed systems are direct responses to one or another of these costs.

Such systems are deployed across a wide range of settings. At one end are large cloud platforms whose nodes occupy geographically dispersed data centers operated by a provider [1], [3]. At the other end are local clusters whose nodes share a single physical location and local network under direct operator control [4]. The two differ in more than scale. A cloud deployment delegates provisioning, maintenance and physical security to the provider. In exchange, the user accepts that data is processed on machines they do not control, that hardware utilization and energy cost are not directly observable, and that all interaction is mediated by the provider's interface. A local cluster reverses each of these: the hardware is fixed but fully visible, data never leaves the operator's premises, and the system can be reshaped at the physical level rather than only through an API. This end-to-end control is what makes a local cluster interesting beyond its raw performance. Properties such as power draw under load, the effect of a configuration change on throughput, or the behavior of the system when a node is added or removed become directly measurable and directly tunable. A local cluster is therefore not simply a smaller cloud but a fundamentally different kind of system, and it is the setting that the remainder of this thesis explores.

2.2 Single-board computers

One option for constructing local clusters is to use SBCs. Unlike conventional computers, which are assembled from separate components, an SBC integrates a processor, memory, storage, and I/O handling onto a single circuit board [5]. Their small physical size (comparable to a credit card) and low power consumption make them attractive building blocks for distributed systems. One of the most widely used SBCs is the Raspberry Pi, which has been used since the early 2010s for a wide array of purposes, including education [5], [6]. SBCs are often relatively cheap, making them a viable choice as cluster nodes. However, their computational performance is limited compared to conventional hardware. Combining many SBCs in a distributed system could offset this limitation while remaining low-cost and energy-efficient [7].

2.3 Node provisioning and shared storage

Before a cluster can distribute workloads, each node must be provisioned with an operating system (OS) and the software required to join the cluster. For smaller local clusters, nodes are usually configured individually by manually installing an OS. This becomes impractical as the number of nodes grows, since each addition requires repeating the same manual steps. One approach is to prepare a *golden image*, a snapshot of a fully configured OS that can be copied directly onto new nodes, making them immediately ready to use. For ongoing configuration changes, *Infrastructure as code* frameworks allow administrators to define system configuration declaratively and apply it to all nodes over the network in a single step [8].

Once nodes are provisioned and connected, a way to exchange data is needed. Workloads often require access to shared input files, and workers need a common location to write their results. Without shared storage, files must be copied to and from each node individually, adding complexity and slowing data distribution. A common solution is a network file system (NFS), which exports a directory from one machine and makes it accessible to other machines on the same network. Nodes read and write to the shared directory as if it were local storage. The trade-off is that all file access passes through the hosting node, which can become a bottleneck if many nodes access it simultaneously or if the hosting node's storage medium is slow.

2.4 Cluster orchestration

Modern distributed systems package applications as *containers*, lightweight isolated environments that bundle an application together with its dependencies and run on a shared host OS. Docker is the most widely used platform for building and running containers [9]. Managing containers across multiple nodes manually is impractical. Orchestration frameworks automate this by deciding which node runs which container, restarting failed workloads, and scaling the number of running instances up or down in response to demand. Kubernetes is the dominant orchestration framework and organizes a cluster around a *control plane*, a designated master node that

coordinates all scheduling and failure management [10] [11].

Kubernetes introduces several abstractions for managing workloads. A *Pod* is the smallest deployable unit, consisting of one or more containers that share storage and network. A *Job* runs one or more pods to completion and tracks their success, making it suited for finite computational tasks. A *DaemonSet* ensures that a copy of a given pod runs on every node in a cluster, which is useful for infrastructure services that must be present on all (or a selection of) nodes, such as monitoring agents. A standard Kubernetes installation carries significant resource overhead, which can be prohibitive on hardware with limited resources. *K3s* is a lightweight distribution of Kubernetes packaged as a single binary and designed for resource-constrained and edge deployments, while remaining fully compatible with the standard Kubernetes API [12].

2.5 Workload background

The cluster is evaluated using three workloads, each chosen to stress a different dimension of distributed performance. Matrix multiplication targets parallel compute throughput across many cores. Password recovery with hashcat targets CPU-bound data parallelism, where each worker performs the same operation on a disjoint slice of the input with no communication between workers. Distributed inference of large language models targets aggregate memory capacity, since the parameter count of modern models routinely exceeds what a single node can hold. The following subsections introduce the concepts needed to interpret the results for each.

2.5.1 Parallelism strategies

When a workload is distributed across multiple machines, the way the work is divided determines both how much speedup is achievable and how much communication is required between workers. Two strategies are relevant to the workloads considered in this thesis.

In data parallelism, the same computation is applied to different parts of the input. The dataset, or in some cases the problem space, is partitioned into disjoint chunks, and each worker processes one chunk independently [13]. Communication between workers is typically limited to distributing input at the start and gathering partial results at the end. Data parallelism scales well when the chunks are roughly equal in cost and when the computation on each chunk does not depend on the others. Both matrix multiplication and password recovery (hashcat) fall into this category.

In model parallelism, a single computation is too large to fit on one worker, so the computation itself is split across workers, each of which holds only a fragment of the overall state [13]. Workers must exchange intermediate values as the computation progresses, which makes model parallelism more sensitive to network latency and bandwidth than data parallelism. This strategy is most often associated with neural network inference, where the parameters of a large model are partitioned across several machines - the approach used for distributed LLM inference in this thesis.

2.5.2 Matrix multiplication

Matrix multiplication is one of the most widely used benchmarks for parallel computation. Its regular structure, well-understood arithmetic intensity, and natural partitioning, make it useful both as a proof of concept for a new cluster and as a basis for measuring sustained throughput. For the square case used in this thesis ($m = n = k = N$), the total number of floating-point operations is approximately $2N^3$ and throughput is reported in GFLOP/s.

Two partitioning strategies are relevant. Row-wise partitioning assigns each worker a contiguous block of rows of A together with the full matrix B, so that each worker produces a horizontal stripe of C independently. Block-wise (or square) partitioning instead divides both A and B into two-dimensional tiles, and each worker computes a tile of C from the corresponding row of tiles of A and column of tiles of B. Row-wise partitioning is simpler to implement and load-balance, while block-wise partitioning can reduce the amount of data each worker must hold in memory at the cost of more intricate scheduling.

2.5.3 Hashcat

Hashcat is a password recovery tool that attempts to identify a plaintext password given its cryptographic hash. The tool computes the hash of each candidate password under the same algorithm used to produce the target, and reports a match when the two hashes match. Hashing is one-way by design, so this brute-force search is, for most algorithms in current use, the only general method available [14].

The work is well suited to data parallelism. The set of candidate passwords can be partitioned arbitrarily across workers, each worker hashes its own chunk independently, and no communication between workers is required until one of them finds a match. There is no shared state, no intermediate synchronization, and no dependency between candidates. This makes password recovery a near-ideal CPU-bound workload for measuring how a cluster scales when communication overhead is removed from the picture.

Hashcat supports several strategies for generating candidates. A mask attack enumerates all strings matching a user-supplied template, such as all strings of a given length drawn from a given character set. The number of candidates grows exponentially with the length of the mask, which makes it difficult to divide the work evenly across workers when shorter and longer masks are mixed. A wordlist attack instead reads candidates from a precomputed list of plausible passwords, often derived from real password leaks. The total number of candidates is fixed and known in advance, which makes it straightforward to assign each worker an equal-sized slice.

2.5.4 Large language models

Large language models are neural networks trained on large text corpora and used to generate or interpret natural language. Their relevance here lies not in their accuracy but in their memory footprint. The size of a model is conventionally reported as

its parameter count, where each parameter is a single learned weight, and inference requires that the parameters be held in memory while the computation runs. Paging parameters from disk on demand is technically possible but too slow to be practical.

A parameter is typically stored in 16 or 32 bits during training, which means a model with several billion parameters requires several gigabytes of memory simply to load, before any activation memory or context buffer is taken into account. Models at the upper end of the current range, with hundreds of billions of parameters, exceed the memory of any single Raspberry Pi 5 by two orders of magnitude.

Quantization reduces the number of bits used to represent each parameter, commonly to 8, 4, or fewer. A 4-bit quantized model occupies roughly a quarter of the memory of the same model at 16 bits, at the cost of a measurable but typically modest reduction in output quality [15]. Quantization makes it feasible to run models on SBC-class hardware that would otherwise be entirely out of reach, but even with aggressive quantization the largest interesting models still exceed the memory of a single node.

The remaining option is to distribute the model itself across several nodes. This is an instance of the model parallelism described in Section 2.5.1, and it is the principal reason LLM inference is included as a workload in this thesis. Where a single Pi cannot hold a large model in memory at any quantization level, a cluster of Pis collectively can. The question of interest then becomes the cost in latency and throughput of doing so.

2.6 Observability

Monitoring the state of a distributed system requires collecting, transmitting and presenting data from every node in the cluster. In the context of this project, observability refers specifically to the real-time and historical collection of hardware and performance metrics, such as CPU temperature, memory usage and processor load, rather than the broader categories of application logs and distributed request tracing.

2.6.1 Collection architectures

Telemetry systems are generally classified as either pull-based or push-based, depending on where the initiative for data transfer lies. In a pull-based architecture, each monitored node exposes an endpoint, typically over HTTP, from which a central server periodically requests data. This is the model used by widely adopted tools such as Prometheus and its ecosystem of per-node exporters such as node_exporter [16]. This central server controls the sampling rate and is responsible for persisting the collected data, usually to a local time-series database. A key assumption of this model is that each monitored node can run a lightweight HTTP server and, in many implementations, buffer recent metrics to local disk between the scrape intervals to avoid data loss.

In a push-based architecture, the monitored nodes themselves transmit data to a central receiver as it is collected, without waiting to be asked. The receiver is responsible for processing and storing incoming stream. This model removes the need for each node to run an HTTP server or to write data to local storage, since metrics leave the node as soon as they are collected. The trade-off is that the receiver must be available to accept data, and any interruption in the connection requires a strategy for handling the gap, whether by buffering on the sender, discarding the data, or pausing collection until the connection is restored.

The choice between these models is shaped by the constraints of the environment, particularly whether nodes have reliable local storage and sufficient resources to run an HTTP server.

2.6.2 Transport, serialization and delivery

The way telemetry is transmitted, encoded and delivered to users affects both the overhead imposed on each node and the responsiveness of the monitoring system as a whole. HTTP is the most common transport for telemetry in pull-based systems, since the central system simply issues GET requests to each node's endpoint. For push-based systems, HTTP can also be used, but alternatives such as gRPC offer advantages in this context. gRPC is a remote procedure call framework built on HTTP/2, which provides persistent connections and bidirectional streaming [17]. A persistent connection avoids repeated TCP handshakes, and streaming allows the sender to transmit messages without waiting for individual responses.

Closely related to the transport is the serialization format used to encode the data. JSON is a widely used text-based format but carries overhead in size and parsing cost. Protocol buffers is a binary alternative in which message structures are defined in a schema file, from which source code is generated for multiple languages [18]. The shared schema eliminates field-name and type mismatches between producer and consumer, and the binary encoding is more compact than JSON.

Once telemetry reaches a central node, it must be delivered to users, typically through a browser-based dashboard. Polling, where the client periodically requests data over HTTP, is simple but introduces delay and unnecessary traffic. WebSockets instead maintain a persistent, bidirectional connection, allowing the server to push new data immediately upon receipt. For a dashboard displaying updates every second, WebSockets are a natural fit.

2.7 Related work

Clusters built from single-board computers have been studied for over a decade, across general-purpose parallel computing, big-data processing, and more recently distributed machine learning. The closest prior work overlaps with parts of this thesis but not the whole, and the differences are noted where they arise.

The general feasibility of Pi clusters for parallel computation is well established.

Gupta et al. ran standard message-passing benchmarks on a Pi cluster and, in doing so, exposed the platform's limits: constrained memory bandwidth prevents all cores from being used effectively, and slow networking restricts the cluster to workloads that are long-running relative to their communication or require little communication at all [19]. This motivates the distinction between communication-light and communication-heavy workloads that informs the choice of evaluation tasks in Section 2.5.

On more recent hardware, Lee and Park built a nine-node Raspberry Pi 5B Hadoop and Spark cluster and evaluated it on terabyte-scale datasets, using the PCIe interface of the Pi 5B to attach NVMe storage and remove the I/O bottleneck of the microSD cards used on earlier models [20]. They found that storage throughput, rather than CPU, limits big-data workloads on this hardware. Their study shares this thesis's platform and its interest in energy efficiency and scaling, but differs in workload and storage: their benchmarks are I/O-bound big-data jobs on PCIe SSDs, whereas the workloads here stress compute, data-parallel throughput, and aggregate memory on the microSD-based configuration described in Section 1.3. On older hardware, Qureshi and Koubaa reached a less favourable conclusion, finding that SBC clusters offered low performance and energy efficiency relative to conventional hardware for the Hadoop workloads they tested [21]. That comparison is not one this thesis contests: performance parity with conventional hardware is not claimed, and the contribution is located instead in practicality, scalability, and the observability tooling described in Section 1.1.

Distributed inference of large language models on commodity clusters is more recent. Prima.cpp is representative: a distributed extension of llama.cpp that runs 30–70B models across a small cluster of everyday devices, partitioning the model so that aggregate memory can hold what no single device could [22]. It is adjacent to the LLM workload of this thesis in its goal, but targets heterogeneous devices over Wi-Fi and optimises for that heterogeneity, whereas this thesis evaluates model parallelism (Section 2.5.1) on a homogeneous, wired Pi 5 cluster.

Purpose-built observability for SBC clusters has received comparatively little attention. General-purpose tools such as Prometheus (Section 2.6.1) assume each node can run an HTTP server and buffer metrics to local disk, assumptions that sit less comfortably on storage-constrained hardware. The lightweight telemetry system described here is intended to address that gap directly.

3. Method

This chapter describes the technical work carried out during the project. The development process was iterative, with decisions frequently revisited as the system took shape and its limitations became clearer. The chapter covers cluster setup and orchestration (visualized in Figure 3.1), the three deployed workloads, the custom observability system, the web-based dashboard, and the benchmarking methodology.

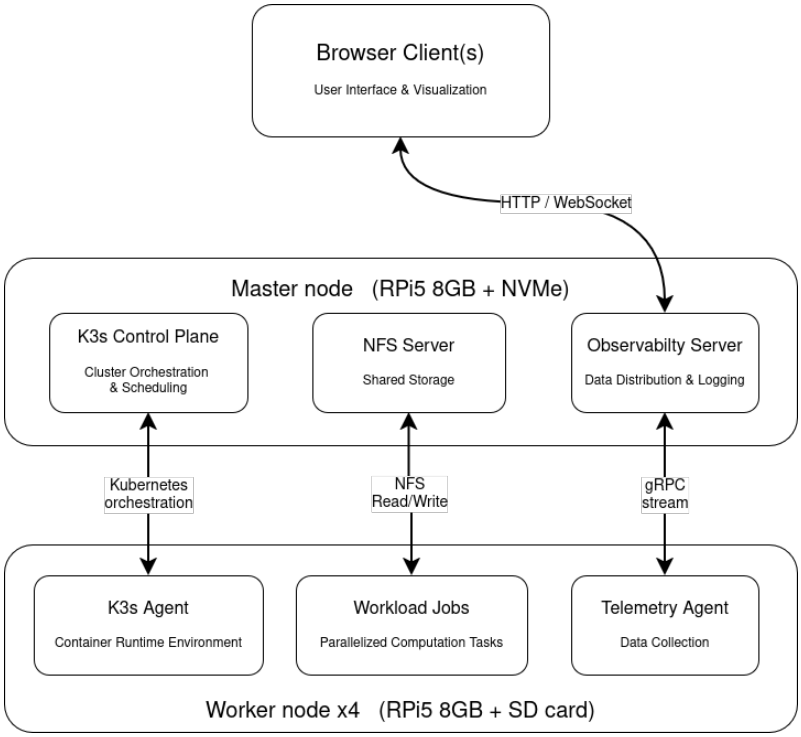


Figure 3.1: Architectural overview of the system.

3.1 Setup

Following section describes the construction of the physical cluster, its configuration for workload deployment, and the reasoning behind architectural choices.

3.1.1 Physical assembly and storage

Physical assembly consisted of installing an NVMe SSD onto the master node for storage, with the worker nodes using SD card. The storage devices were benchmarked to ensure that they were working correctly. An NFS was set up to ensure that all nodes have access to a storage space over LAN (Local Area Network) in order to achieve a simpler workflow, removing the need to send files to each individual node. This was done by designating a partition of the master nodes SSD as an NFS share, then mounting the path to the NFS on each node so they can access it

through LAN. This would also alleviate the minimal 32 GB storage of the SD cards compared to the 1 TB of the SSD, allowing the SD cards to only store the essential operating system files.

Figure 3.2: Photo of the physical hardware.

3.1.2 Operating system provisioning

The operating systems were initially installed and configured manually one-by-one, but a golden image was later set up to simplify the process. A golden image is essentially a snapshot, or image, of a freshly installed operating system that has been properly configured. This means that once the image is copied on to the SD card, the node is ready for usage. The golden image later caused several problems due to a configuration error, leading to its removal and having to install nodes manually again. This is still practically acceptable as it is ideally only needed once.

3.1.3 Cluster orchestration

To install the orchestration software of choice, Kubernetes, onto the nodes, Ansible was used. Ansible is an automation tool that uses playbooks to efficiently automatize this process. A playbook is a file that defines repeatable, declarative automation tasks. By setting up an Ansible playbook with the details of the nodes, such as hostname, ip address and password, Kubernetes can be installed over local connection to all nodes simultaneously. Other playbooks were also used, mainly the built-in reset playbook to quickly re-install Kubernetes in case of possible node failure.

3.2 Workload deployment

Three workloads were chosen to run on the cluster; matrix multiplication, hashcat, and distributed LLM inference. Matrix multiplication was the initial workload

deployed, due to its simplicity serving as a proof of concept of parallelizable performance, thus achieving lower compute times with multiple nodes. A simple workload was deemed necessary to start out with before moving on to more complex ones, as managing and maneuvering Kubernetes was non-trivial. Hashcat and LLM inference showcased real-world use cases, those being CPU- and RAM-intensive tasks respectively. Significant speedups are achieved in hashcat through data parallelism, while larger, and thus smarter models, can be deployed with additional RAM.

The method of deploying workloads had to be kept simple, as extending the number of cases had to be kept in mind. This was achieved through a python script running on the master node, which would scan a folder for input files. Based on filename, e.g. matrix.txt, the script would utilize the Kubernetes API (Application Programming Interface), and the functions it provides.

First off, the clusters hardware is scanned and saved. After the hardware is measured, a job specification is created. A specification requires file paths, variables, the code necessary for the work that is to be deployed, and the earlier measured hardware specifications. This specification is then used when submitting a job to Kubernetes. After a job is submitted, the script will wait for the job to be reported as finished. When this happens, it will gather the results that the pods have placed into the NFS, and join them into a final result. Once this is done, the result is accessible to the end-user and the NFS is cleaned up. The script then resumes scanning for additional input files, ready to deploy another workload.

3.2.1 Matrix multiplication

The workload consisted of multiplying two dense square matrices generated dynamically within each worker pod using the NumPy function `np.random.rand()`. The matrices contained random floating point values uniformly distributed between 0 and 1. A fixed random seed was also used to ensure reproducibility between executions. Matrix generation was performed locally within each worker pod as part of the containerized workload implementation. Since each pod executed an independent container image, generating the matrices internally simplified workload distribution and reduced the need for transferring large datasets or dynamically modifying workload data from the master node.

Two workload partitioning strategies were implemented: row-wise, and square-wise partitioning. In the row-wise implementation, the input matrix was divided into horizontal row segments, where each worker pod computed a subset of the resulting matrix rows. With this solution each pod only required access to its assigned row range and the full second matrix. An alternative square-wise implementation was also developed, where the matrices were partitioned into smaller sub matrices. Unlike the row-wise approach, this introduced a two dimensional component to the workload, requiring stricter policies when splitting the matrix.

3.2.2 Hashcat

One of the selected workloads was password recovery using hashcat. This was chosen because it supports a form of data parallelism that is relatively straightforward to implement. The workload can be divided into independent parts, allowing multiple worker nodes to process different sections of the search space at the same time. This also makes it possible to achieve a fairly even load across the worker nodes during runtime.

The first attack method considered was a mask attack. This method was initially chosen because it was simple to set up and had a conceptually clear way of dividing work between pods. For example, one pod could test all password candidates of length two, another pod all candidates of length three, and so on. However, this approach resulted in poor load balancing. The number of possible candidates increases rapidly as the password length increases. This means that a pod testing shorter passwords would finish much earlier than a pod testing longer passwords. As a result, the worker nodes would not be utilized evenly, which would reduce the efficiency of the parallel execution.

For this reason, the attack method was switched to wordlist attack. With a wordlist attack, each worker node can be assigned a separate section of the wordlist. It is therefore easier to ensure a good workload balance between the nodes. The RockYou wordlist was chosen because it is a widely used dataset for testing wordlist-based password recovery.

A python script was written to be distributed to each Kubernetes pod with each pod being sent their own index. This was tested locally to confirm its functionality. The pods were simulated by running their script locally on separate terminals with each terminal running the script on a chunk of the total wordlist.

Before benchmarking, the wordlist needed to be made available to the worker nodes. One alternative was to store the wordlist on the NFS and let each worker read from it during execution. This would reduce local storage usage on each worker, but it could also increase network communication and make the shared file system a bottleneck. The second alternative was to provide each worker with a local copy of the wordlist and the hashes to decrypt through the docker image. This reduces dependency on the NFS during the process.

For the benchmarks, the workload configuration utilized MD5-crypt with a salt. This produced a longer execution time and made the measured speedup more representative of the actual hashcat workload. The benchmark used a worst-case scenario for a single salted MD5-crypt hash, where the correct password chosen was not in the wordlist. This ensured that the workers had to process the full assigned workload before a result was found. The hashcat command was also run with the `-w 3` workload profile. This setting was used to increase hashcat's workload intensity and improve performance, while keeping the configuration consistent across all benchmark runs.

3.2.3 Distributed LLM inference

To minimize setup complexity, llama.cpp was built only on the master node, and stored on the NFS together with the models. The deployed models were Qwen3 models of varying sizes. Multiple models were selected to evaluate different deployment scenarios. First, smaller models that could fit on a single Pi were tested to determine whether distributed execution could still provide performance improvements. Offloading inference from the master node to a separate node could improve performance, by reducing computational load on the master node. The second scenario involved testing a significantly larger model compared to the one used for performance benchmarks, a model that could not fit on a single node. Due to the way llama.cpp distributes computation, models of sufficient size can place only minimal workload on the master node while distributing most of the computation across the worker nodes.

3.3 Observability

This section describes the observability system developed for the cluster. The system collects hardware and performance data from each node, making it available both for the benchmarking analysis presented later and for real-time visualization through the dashboard.

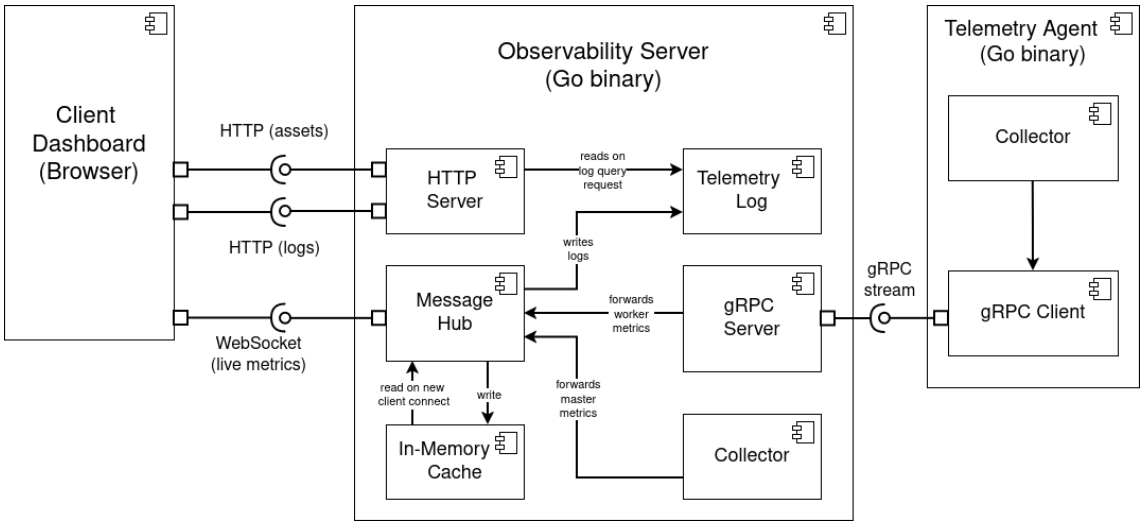


Figure 3.3: Component diagram for the observability system.

3.3.1 Architecture and rationale

While several mature open source solutions exist for collecting hardware and performance data from a cluster, a bespoke implementation was chosen for three main reasons. First, full control over which metrics are collected, how they are formatted, and how frequently they are reported was considered important. Second, a system designed in-house would be sufficiently lightweight and transparent to be debugged and extended, avoiding the risk of subtle performance overhead or hidden behavior

in a larger third-party tool. Third, as the worker nodes run from SD cards, it was important that the telemetry pipeline impose minimal I/O load and that no metric data be written to local storage.

As discussed in Section 2.6.1, pull-based telemetry systems assume that each node can buffer data to local disk and run an HTTP server. Neither assumption holds well for SD-card-based worker nodes, so a push-based architecture was chosen. Each worker collects its metrics in memory and immediately transmits them to the master node over gRPC, with no intermediate disk storage and no locally running HTTP server on the worker.

The master node is responsible for receiving, processing, and persisting the reports, and for making them available to clients through two interfaces: a WebSocket stream that delivers live data to connected clients, and an HTTP endpoint through which clients may request historical data for a specified time range. Persistence is implemented as append-only JSON Lines files on the master's NVMe SSD, rotated once per day, avoiding the need for a database. As a result, worker nodes only bear the cost of collecting data, while the master handles all downstream buffering, storage and delivery to clients.

The overall structure of the observability system is illustrated in Figure 3.3, which shows the components described in the following subsections and the interactions between them.

3.3.2 Shared collector and data model

Because telemetry is required from both the master and the worker nodes, a single collector module was implemented in Go and reused on both node types. The worker transmits the data it produces to the master over gRPC, whereas the master passes its own data directly into the processing and storage layer within the same process, avoiding a redundant network hop to itself. This arrangement reuses the collection code across both node types while eliminating unnecessary networking overhead on the master, and provides a single source of truth for how metrics are gathered and formatted. As a consequence, changes to the metric schema or to the collection logic only need to be made in one place, which simplifies both maintenance and future extension of the system.

The collector was implemented in Go for its execution speed, low memory footprint, and straightforward concurrency primitives, all suited to the requirement that collection on the worker nodes remain inexpensive. Go's ecosystem around gRPC and cluster tooling is also mature. All telemetry messages were defined as Protocol Buffers schemas (Section 2.6.2), from which both the Go types used by the master and worker and the TypeScript types used by the client were generated.

3.3.3 Ingress, resilience and reconnection

On the master, the gRPC ingress and HTTP server listen on separate ports within the same process, so that a single master binary exposes both the worker-facing and

the client-facing interfaces. On the worker, telemetry is held only in memory and is never buffered to the SD card: if the gRPC stream to the master becomes unavailable, the worker pauses collection entirely and enters a reconnect loop with exponential backoff, resuming collection only once the stream has been re-established, as shown in Figure 3.4. This was a deliberate trade-off: buffering locally would reintroduce the SD-card writes the design set out to avoid, and collecting into memory during an outage would spend CPU cycles on data that could not be delivered. On reconnect, the worker immediately transmits a full report containing static, config and dynamic sections, which allows the master to rebuild its per-node view without waiting for the slower static and config intervals to elapse.

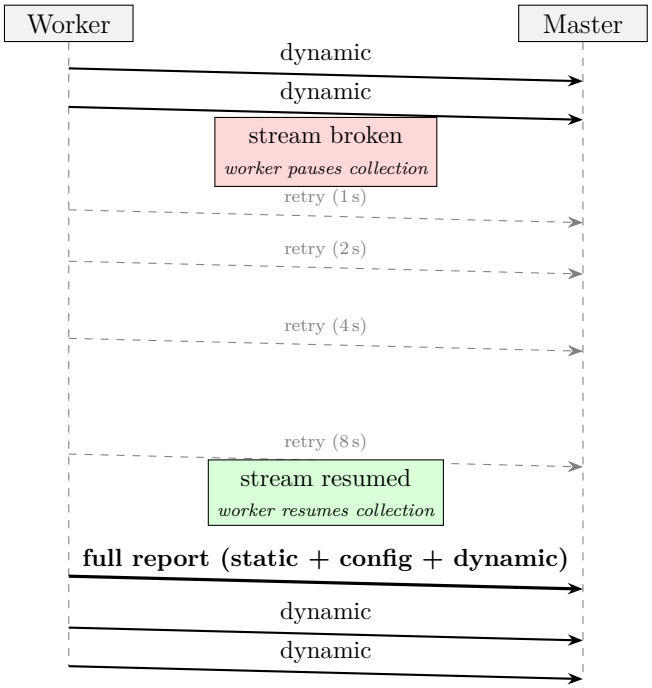


Figure 3.4: After a stream failure, the worker enters an exponential-backoff retry loop (dashed) and on reconnect, transmits a full report containing all three telemetry subtypes before resuming the normal cadence.

3.3.4 Client and HTTP server

The client interface was implemented as a single page application (SPA) using Svelte, a JavaScript user-interface framework that compiles components to self-contained modules at build time rather than shipping a runtime framework and a virtual document object model (DOM) to the browser. This results in smaller bundles and lower CPU and memory usage on the client a desirable property given the continuous stream of telemetry updates. The compiled assets were bundled alongside the Go binary and served by the same HTTP server, so no separate frontend service needs to be deployed or kept in sync with the backend. The full set of endpoints is listed in Table 3.1.

3.3.5 Collected metrics

The metrics selected for collection describe the hardware specification, the current configuration, and the real-time performance of each node. Each node is identified by

Table 3.1: HTTP and WebSocket endpoints exposed by the master node.

Path	Protocol	Purpose
`/`	HTTP GET	Serves the static assets of the dashboard.
`/ws`	WebSocket	Pushes live telemetry to the client.
`/api/logs/list`	HTTP GET	Returns the set of available daily log files, each annotated with the date and the local-time bounds of the first and last record.
`/api/logs/raw`	HTTP GET	Streams the raw JSONL records from a given days log file, optionally filtered by a start and end time.

its hostname. Additional network metrics include the primary internal IP address, a connectivity flag, and the list of network interfaces with their names and MAC addresses. The CPU is characterized by the core count, temperature, frequency, frequency bounds and the active and supported governors. Memory and storage are described by installed, free, and available capacities. Real-time load is captured by the system load averages over one, five, and fifteen minutes, together with raw CPU counters (user, system, idle, I/O wait and context switches) accumulated since boot, allowing derived quantities such as utilization to be computed on the client side over arbitrary time intervals. Each report also carries three high-precision Unix timestamps marking when the worker transmitted the report, when the master received it, and when the master forwarded it to clients, from which latency can be computed. Finally each report also records the collection duration in nanoseconds and an error-flag bitmask, where zero indicates a healthy report.

Table 3.2: Telemetry metrics grouped by report type and transmission cadence.

Static (60 s)	Config (30 s)	Dynamic (1 s)	Every report
Nr of CPU cores	Hostname	CPU frequency	Logging ts
Network interfaces (names, MACs)	Primary internal IP address	CPU temperature	Sender hostname
Total RAM	Web access flag	Free and available RAM	Worker transmit ts
Storage capacity	Active CPU governor	Remaining storage	Master receive ts
Available CPU governors	Min/max CPU frequencies	CPU counters (user, system, idle, iowait, context switches)	Master forward ts
	Load avg (1/5/15m)	Collection duration	Error flag bitmask

3.3.6 Report types and cadence

The metrics were divided into three report types. Static reports contain information that is effectively constant for the lifetime of a node, such as the number of CPU cores, the network interfaces, the total installed memory and storage, and the set of available governors. Config reports contain information that changes infrequently but is not fixed, such as the hostname, the active governor, the minimum and maximum CPU frequencies, the primary IP address, and the system load averages. Dynamic reports contain information that changes on short timescales, such as the current CPU frequency, the current CPU temperature, the amount of free and available memory, the remaining storage, and the raw CPU counters. In the deployed configuration, dynamic reports are transmitted once per second, config reports every thirty seconds, and static reports every sixty seconds. Static reports are therefore transmitted rarely, config reports less frequently than dynamic reports, and dynamic reports at the highest rate. The complete set of metrics and their assignment to report types is summarized in Table 3.2. This layering reduces the bandwidth used and the work performed on each node while ensuring that fast-changing values remain current in the visualization and in the historical record.

This layering also shapes how the master serves connecting clients. The master caches the most recent static and config payloads per node in memory, so that a newly connected dashboard receives the current hardware description and configuration immediately rather than waiting for the next report interval. Dynamic reports are excluded from this cache as they are high-frequency and short-lived, so clients instead receive them live.

3.3.7 Packaging and deployment

Both the master and worker components were packaged as Docker images and stored alongside the rest of the project in a monorepo. A GitLab CI pipeline, executed on a self-hosted runner, built the images for the ARM architecture required by the Pis and published them to the project registry. The images were deployed on the cluster using Kubernetes DaemonSets, with node selectors distinguishing the master component from the worker component, so that the correct container was pulled and started exactly once on each node of the appropriate type. This meant that the telemetry system was tied to the lifecycle of the cluster rather than to individual jobs, so any node that joined the cluster would automatically begin reporting without further configuration.

3.4 Dashboard

The dashboard is the interface through which a user interacts with the cluster. It runs in any modern web browser and is served by the master node. From it, the user can inspect per-node and cluster-wide metrics in real time, review historical telemetry logs, and monitor node connectivity. The visual style was designed to suggest a system-level tool rather than a consumer web application, drawing on

the look of older terminal interfaces and BIOS menus. This is expressed through a dark blue background, white panel borders, and a mono-space typeface to form the core of the design. Yellow and green are used sparingly to highlight important values and healthy states. Styling uses Tailwind CSS with a shared set of theme tokens, so that the color palette is defined once and reused across all components. Color is used sparingly on the NODES tab to draw attention without dominating the layout. Values that warrant operator attention, such as memory utilization above an internal threshold, are rendered in a contrasting color, while values in their healthy range are rendered in green. The master node's hostname is highlighted to make it identifiable at a glance when scanning a grid of otherwise similar cards.

The dashboard is a single-page application. Six tabs sit at the top of the screen, of which four are functional: MAIN, NODES, LOG, and GRAPH. The remaining two, WORKLOADS and TERMINAL, are discussed in Section 5.3. The MAIN tab gives a cluster-wide overview, showing how many nodes are online and a summary of aggregate resource use. NODES presents one card per connected node in a two-column grid, making it possible to compare nodes side by side. LOG surfaces diagnostic information from the telemetry stream itself, such as the error flags reported by the collector when a metric fails to read, allowing the user to confirm that telemetry is being gathered cleanly across the cluster. GRAPH plots rolling windows of live telemetry as time-series using *lightweight-charts*, chosen for its small bundle size and minimal runtime overhead.

The dashboard maintains a single in-memory map keyed by node hostname. When a message arrives, the relevant fields on the matching node are updated while everything else is preserved. This means a slow-changing static report and a fast-changing dynamic report can both contribute to the same node entry without overwriting each other. Removing a node from the screen the moment it stops reporting was tried during early development and found to be more confusing than helpful, so offline nodes are instead kept visible in their last known state, dimmed and shown with a label indicating how long ago they were last heard from. The list of nodes is sorted so the master appears first, followed by online workers in alphabetical order, with offline workers placed at the bottom.

3.5 Benchmarking

The system is evaluated quantitatively against a set of metrics chosen to characterize its performance, scalability, efficiency and orchestration overhead. Computational performance is measured through speedup tests comparing single-node execution against multi-node configurations. For matrix multiplication, performance is measured in GFLOPS. The single-node execution serves as the baseline, and the performance of each multi-node configuration is expressed as a speedup factor relative to this baseline. This speedup factor is the measurement used for hashcat.

For LLM inference, performance is measured in tokens per second. Speedup experiments use the Qwen3-4B model, which fits on a single node and therefore allows direct comparison between single-node and distributed execution. Memory scala-

bility is evaluated separately using the Qwen3-32B-Q6_K model. This model is quantized in order to enable execution across multiple nodes while still exceeding the memory capacity of a single node.

Thermal and power efficiency are also examined, since Raspberry Pi systems are susceptible to thermal throttling under sustained workloads. The relationship between operating temperature, workload intensity, and clock frequency is analyzed to determine when throttling occurs and how it affects sustained performance. Energy consumption is measured in watts using a PeakTech 9035 inline power meter connected between the cluster and the wall outlet. Because the meter reports only instantaneous power draw, total energy consumption for a workload is estimated by multiplying the measured power by the runtime of the application.

A runtime breakdown is recorded for each deployment in order to identify orchestration and deployment bottlenecks. The measured phases first include hardware surveying, job specification creation and submission, container building and distribution. After this, application execution is measured. Finally, result handling through transfer to the master node and final aggregation, as well as NFS cleanup. Potential bottlenecks examined include the limited storage performance of the SD cards on the worker nodes, together with thermal throttling and orchestration overhead as mentioned above.

4. Results

This chapter presents the results of the project. Sections 4.1, 4.2, and 4.3 present the performance, scaling, and energy results for matrix multiplication, hashcat, and LLM inference respectively. Section 4.4 evaluates the custom telemetry system, covering collection overhead, thermal behavior, and an unplanned hardware discovery. Section 4.5 summarizes the dashboard implementation.

4.1 Matrix multiplication

This section evaluates the scalability and efficiency of distributed matrix multiplication on the Kubernetes cluster. Performance was analyzed by varying the number of pods, nodes, CPU frequencies, and partitioning strategies. The results highlight the trade-offs between parallel scalability, communication overhead, resource contention, and energy efficiency.

4.1.1 Performance and scaling

Speedup is measured from the aggregate compute throughput of the cluster in GFLOP/s for the compute performance. For each configuration, the total cluster throughput was computed by summing the measured GFLOP/s throughput of all active worker pods to represent total cluster performance. The tests were done with factors beyond the pods count being fixed, those include handling 8192 x 8192 matrices and performing 10 runs to acquire an average performance over the runs. The results of the runs can be observed in figure 4.1.

Pods represent the total number of parallel workers used in the cluster. In the 4-pod configuration, each node executes a single pod, resulting in a fully distributed baseline setup.

Speedup is computed relative to the 4-pod configuration, which is defined as the baseline, 1x. Remaining configurations are normalized against the baseline reference.

Pods are total for the cluster. Consequently, in the case of four pods they are divided equally on the cluster, resulting in one pod per node for this configuration. The most significant improvement was observed when scaling from four to eight pods, resulting in a speedup of 1.68x. This demonstrates that the matrix multiplication benefited substantially from parallelization. However, the achieved speedup remained below ideal linear scaling, and further increasing the number of pods did not improve GFLOP/s performance.

During testing, it was observed that Kubernetes consistently attempted to utilize all available CPU resources regardless of the number of scheduled pods. As the number of pods increased, the workload likely experienced greater contention for shared hardware, primarily CPU. Additional overhead from Kubernetes scheduling,

thread management, and synchronization may also have contributed to the reduced scaling efficiency. Consequently, increasing the number of pods beyond a certain point resulted in diminishing returns and a slight decrease in overall GFLOP/s performance.

One possible explanation for the limited scaling behavior is the interaction between NumPy's internal parallelization and Kubernetes resource scheduling. The matrix multiplication implemented through `numpy.dot()` relies on optimized linear algebra libraries that are capable of multithreaded execution. As a result, each pod may already have utilized a degree of parallel execution. However, the effectiveness of this internal multithreading may have been constrained by Kubernetes scheduling and resource allocation policies. Consequently, the available CPU resources may not have been consistently accessible to each pod during execution. Fully characterizing this behavior was outside the scope of this project, as there is limited visibility regarding Kubernetes runtime scheduling and thread allocation.

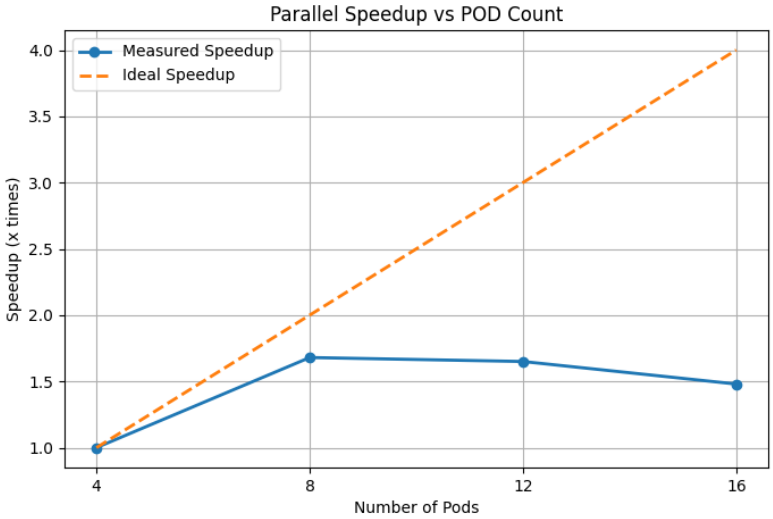


Figure 4.1: Speedup of distributed matrix multiplication vs pod count, 4-pod baseline.

With the optimal number of pods tested, resulting in 2 for each node this configuration was used to test the nodes scaling which can be seen in table 4.1. The test was done using 8192 x 8192 matrices and the results were taken from an average of 10 iterations.

When increasing the number of nodes, it scales close to linear with the hardware as opposed to when increasing the pods per node. Therefore, it appears to be mostly scheduling issues with software that have an impact on the parallelization opportunities.

Nodes	Performance [GFLOP/s]	Speedup (x times)
1	36,37	1
2	71,37	1,96
3	105,84	2,91
4	137,84	3,79

Table 4.1: Performance scaling of distributed 8192×8192 matrix multiplication across nodes. 3 pods, row-wise partitioning.

In figure 4.2 a visualization is presented to illustrate the extent to which near ideal linear scaling was achieved. The graph compares the measured speedup of the distributed matrix multiplication system with the ideal linear scaling baseline. It was performed on 8192 x 8192 matrices with 8 pods and iterated 10 times for a result average. 8 pods was chosen after it showed the best performance in figure 4.1 The percentage values indicate how close the system is to perfect scaling as the number of nodes increases. However, this only accounts for the compute power.

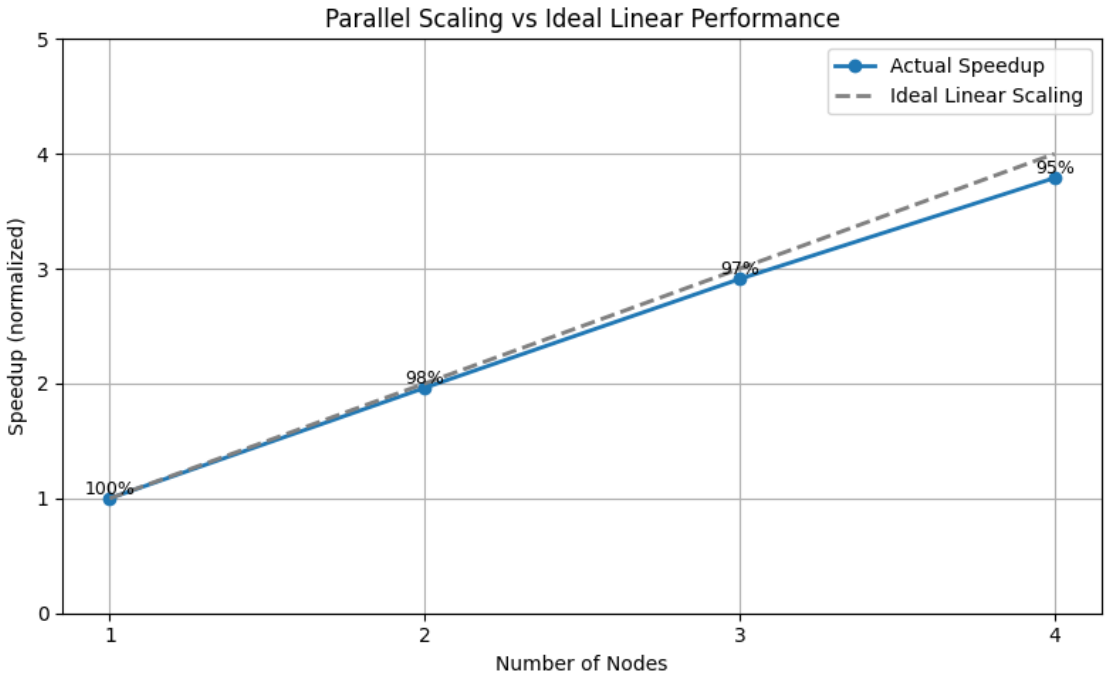


Figure 4.2: Measured parallel speedup compared to ideal linear scaling for distributed matrix multiplication.

4.1.2 Energy efficiency

Figure 4.3 presents the results of 10 iterations of 8192 x 8192 matrix multiplication performed at each CPU clock frequency available, using increments of 100Mhz.

While increasing frequency results in higher GFLOP/s, the marginal performance gains diminish at higher frequencies. An explanation may be that a shift in bot-

tleneck happens from compute to memory and communication overhead. However, this could also be due to increase in thermal throttling as more power is used, resulting in the clock frequency being lowered by the Pi. Reaching this threshold for throttling could result in a drastic change in clock frequency rather than it being dynamically regulated, which would incur a significant performance drop.

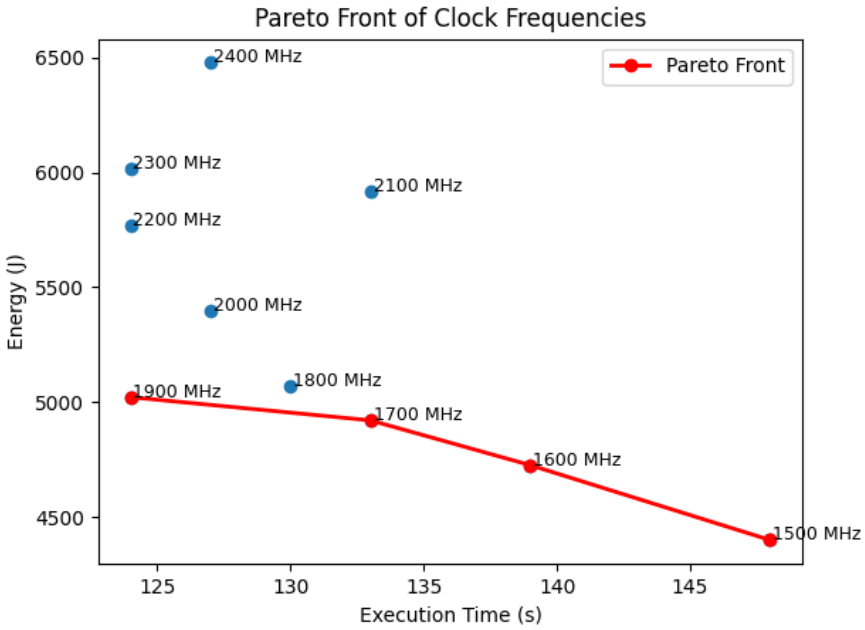


Figure 4.3: The Pareto plot illustrates the trade-off between computational throughput and resource efficiency across different clock speeds.

Performance optimization can be achieved by increasing both available hardware resources and improving how the workload is distributed across those resources. With this in mind, a square decomposition strategy was evaluated in addition to the standard row-wise decomposition method. The performance difference can be seen in table 4.2. The times for compute and save are the average over 10 iterations of 8192 x 8192 matrix multiplication. The total time is also the average over 10 iterations, it is the total time of the program to execute. Hence it sums to more than the compute and save together, as this includes overheads.

When attempting to run the square decomposition at first with 1024MB RAM per pod it crashed, therefore four pods per node was not possible to run with the memory available. Instead it had to be lowered to two pods per node, therefore the square implementation reaches a bottleneck by RAM before the row implementation does. On the other hand, this square implementation was also done in python, like the row implementation. When optimizing memory it could instead be beneficial to use another language with more optimization that can control the hardware on a lower level.

The square decomposition achieved performance comparable to the four pod row-

wise implementation. However, the square approach imposes stricter scalability constraints, since the workload must be divided into a square number of partitions. As a result, the number of pods must form a perfect square, such as 4,9 or 16. In contrast, the row-wise implementation allows more flexible scaling with arbitrary pod counts.

The average compute performance was slightly higher for the row-wise implementation, indicating more efficient computation under the tested configuration. However, the total runtime slightly favored the square decomposition. This difference is likely related to the communication overhead rather than the computational performance itself. The two decomposition strategies distribute matrix data differently between workers, which affects both the amount and pattern of communicated data during execution. Additionally, the scaling behavior may vary depending on the matrix size used, meaning that different matrix dimensions could produce different performance characteristics.

Splitting	**Compute Average [s]**	**Save Average [s]**	**Total [s]**
Row	13,22	2,88	18,4
Square	13,25	2,4	18,1

Table 4.2: Comparison of row-wise and block-wise (square) matrix multiplication implementations showing average compute time, save time, and total execution time.

4.2 Hashcat

This section describes the results gathered from benchmarking through various setups for hashcat. The results and reflections focus on the performance and the energy usage while running the tool.

4.2.1 Performance and scaling

The hashcat workload was benchmarked using different numbers of nodes and pods per node. Table 4.3 shows the results when running one pod per node but varying the number of nodes. Each test was done 5 times to decrease the effect of outliers. Both the total execution time and pod execution time decreased as the number of nodes increased. This shows that the workload scaled well when distributed across more nodes.

The speedup values in the result tables were calculated from the measured benchmark runtimes. The configuration with one node and one pod per node was used as the baseline for node scaling, while the configuration with four nodes and one pod per node was used as the baseline for the pods-per-node comparison. Speedup was calculated with the following formula.

$$S = \frac{T_{\text{baseline}}}{T_{\text{configuration}}}$$

Here T_{baseline} is the runtime of the baseline configuration and $T_{\text{configuration}}$ is the runtime of the tested configuration.

Nodes	Pods / Node	Total Time (s)	Pod Time (s)	Speedup
1	1	1725.25	1716.25	1
2	1	897.89	889.23	1.92
3	1	621.66	612.98	2.78
4	1	481.34	474.93	3.63

Table 4.3: Hashcat benchmark results for MD5-Crypt using one pod per node. Pod time is time for only the hashcat script to run.

Table 4.4 shows the effect of increasing the number of pods per node on a 4 node setup. The baseline was the configuration of four nodes and one pod per node. Just like previous tests, the results are a mean out of 5 tests.

The best result was achieved with two pods per node, which achieved a speedup of approximately 1.69x.

Pods / Node	Total Time (s)	Pod Time (s)	Total Speedup
1	481.34	474.93	1.00
2	282.74	275.22	1.70
3	290.81	281.49	1.66
4	302.85	292.47	1.59

Table 4.4: Hashcat speedup when increasing the number of pods per node on four nodes.

Overall, the results show that increasing the number of nodes improved the performance significantly. Increasing the number of pods per node also improved performance but only up to two per node.

4.2.2 Energy efficiency

The additional energy usage of the hashcat workload was estimated by subtracting the baseline power draw from the measured workload power. This was done to isolate the power draw caused by the benchmark itself, excluding the power that the cluster consumed while idle. Note that the workload had a time in the beginning for each run of about 69 seconds where power draw was slightly lower. This is factored into the final calculations.

$$E_{\text{additional}} = \frac{(P_{\text{workload}} - P_{\text{base}}) \cdot t}{3600}$$

Table 4.5 shows the estimated additional energy usage when increasing the number of nodes with one pod per node. The additional power draw increased as more nodes were used, from 1.40 W with one node to 11.60 W with four nodes. However, the total execution time decreased as the number of nodes increased.

Table 4.6 shows the estimated additional energy usage when increasing the number of pods per node on four nodes. The additional energy usage was lowest with one pod per node when only energy is considered, but two pods per node achieved nearly the same additional energy usage while substantially reducing the execution time. The three and four pod configurations did not improve the runtime further and consumed slightly more additional energy than the two pod configuration. During the two pod benchmark runs, telemetry showed that the CPU cores were already operating at full load. This suggests that additional pods mainly introduced resource contention rather than useful parallel execution.

Nodes	Total Time (s)	Workload Power (W)	Energy (Wh)
1	1725.25	1.4	0.65
2	897.89	4.6	1.12
3	621.66	8.4	1.4
4	481.34	11.6	1.49

Table 4.5: Estimated energy usage for Hashcat when increasing the number of nodes with one pod per node.

Pods / Node	Total Time (s)	Workload Power (W)	Energy (Wh)
1	481.34	11.60	1.49
2	282.74	20.20	1.47
3	290.81	20.30	1.52
4	302.85	20.60	1.61

Table 4.6: Estimated additional energy usage for hashcat when increasing the number of pods per node on four nodes.

4.3 LLM inference

This section evaluates the scalability and performance of distributed LLM inference on the cluster. Performance was tested by using a 4B parameter model that could fit on any of the nodes on it's own to give a fair comparison. Scalability of LLM inference was tested using a much larger 32B parameter model which was only able to fit in memory when distributed across a cluster of at least 4 nodes. The results give insight into how distributed LLM inference works on llama.cpp and highlights both pros and cons of this cluster for similar workloads.

4.3.1 Performance

Computational speedup is measured by running the same prompt several times on different cluster configurations. Due to the way LLMs work the work done is non-deterministic. Therefore, tokens/s is used for performance measurements. Additionally, the result is an average after 5 runs to minimize the difference.

Performance for the cluster during LLM inference using the prompt "Benchmark" listed in 4.7. The prompt typically resulted in the model generating an information text about benchmarking and its purposes. Reasoning was turned off for all tests due to time constraints during benchmarking. No speedup was measured in these tests. The same model was used for all tests. All models started inference at the peak speed, and degraded as the context window grew.

Nodes	Tokens	Time	tokens/s	Peak tokens/s
1	820	307s	2.67	3.3
2	762	283s	2.69	3.4
3	772	287s	2.69	3.3
4	780	290s	2.69	3.4

Table 4.7: Performance scaling of distributed LLM inference across nodes.

4.3.2 Memory scaling

Memory scaling is measured using a much larger LLM compared to the model used for measuring speed. A 32B-Q_6 model was used, representing a total size of 26.2GB. Running the full model required 26,378 MiB of memory across all worker nodes. Additionally 604MiB of memory was used on the master node in order to handle model logic. Loading the model onto the cluster took a sizable amount of time seen in 4.8. Tensor loading is done on the master node. However model loading was much faster when the model was cached also seen in 4.8 suggesting benefit from OS-level file caching. Notably this transfer is much slower than the networking speed of the cluster. Transferring 26,378 MiB over 860 seconds is a rough transfer speed of 30 MiB/s. Compared to the 1 Gigabit connection of the network, around 120 MiB/s, this is quite slow. This was backed up by looking at the rate of how quickly memory was being occupied in the telemetry system.

Stage	Elapsed time
Tensor loading (initial)	56 seconds
Worker 1 fully loaded	269 seconds
Worker 2 fully loaded	476 seconds
Worker 3 fully loaded	667 seconds
Full cluster ready (cold)	860 seconds
Full cluster ready (cached)	323 seconds

Table 4.8: Time to load the Qwen32B-Q_6 model onto 4 worker nodes.

This model is large enough to fit fully in RAM on the full cluster of 4 nodes without the master having to put in much work. When run on less than 4 nodes llama.cpp distributes a significant amount of work on the master node. This is undesirable in our cluster as the master is occupied with other tasks. If the master node is occupied with running inference this could interfere with tasks such as telemetry.

Additionally if less than 3 nodes are used the master must also handle swapping tensors between RAM and disk which impacts performance massively.

Once the large model is fully loaded onto the cluster inference throughput was roughly 0.4 tokens/s. No difference was measured when reasoning was enabled or disabled.

4.3.3 Power draw

During benchmarking of the 4B parameter model power draw was also measured, and displays an interesting part of how llama.cpp functions. For the power draw tests the four worker nodes were all turned on at the same time, but LLM inference was only run on a certain number of nodes at a time.

Table 4.9: Power draw during LLM inference on the cluster

Nodes	Power draw
0 (Idle)	14W
1	20W
2	21W
3	22W
4	24W

As can be seen in 4.9 power draw remains roughly the same throughout the tests. The power draw remains the same with 4 nodes when using both models tested. To add on to this, data on temperatures was only measured when using 1 node and 4 nodes, yet displays a strength of clustering, being thermal distribution. When using a singular node the processor stayed around 77°C. When running all 4 the average temperature was 61.75°C. Even though the 4B parameter model performed the same no matter the amount of nodes used, when clustering, the processors were cooler compared to when using a singular node.

4.4 Observability

This section evaluates the custom telemetry system described in Section 3.3. We first characterize the collection overhead imposed on worker nodes in Section 4.4.1, then examine the thermal behavior of the cluster under load and the role of the telemetry system in identifying performance-relevant hardware differences in Section 4.4.2.

To evaluate the telemetry system, eleven runs of the same matrix multiplication workload were performed under varying conditions. The workload used 8192×8192 matrices distributed across four nodes with three pods per node. The test configurations are summarized in Table 4.10. Each row represents a single job submission, where each pod internally performed 10 iterations of the multiplication. Tests 1–5 ran with telemetry enabled; of these, Tests 3–5 additionally connected an increasing number of dashboard clients (one, four, and eight respectively) to determine whether serving live data to browsers introduces overhead on the worker nodes. Tests 6.1–6.3

ran with telemetry disabled entirely, providing a no-telemetry baseline against which the overhead of the telemetry DaemonSets can be assessed. Because telemetry was disabled during these runs, the wall-clock duration is the only measurement available for comparison. Tests 7.1–7.3 ran with telemetry enabled but with the cooling fans physically disconnected, in order to provoke thermal throttling and observe it through the collected data. All other parameters remained unchanged. The *Duration* column records the wall-clock time from job submission to completion. The *Gap* column records the idle interval between runs, since as the fan-removal tests suggests, insufficient cooling leads to progressively higher starting temperatures and longer execution times.

Each test configuration was submitted as a single Kubernetes Job, so the durations in Tables 4.10 and 4.12 each represent a single measurement encompassing all 10 iterations plus orchestration overhead. The variability is visible in Tests 6.1–6.3, which share identical configurations yet span 121–139s. This 18s spread sets a floor on measurement uncertainty and should be kept in mind when interpreting small differences elsewhere.

Table 4.10:
Test configurations for the observability evaluation. Each row is a single submission of the same 8192×8192 matrix multiplication workload on four nodes with three pods per node, where each pod internally performed 10 iterations. *D*: wall-clock duration. *T*: whether the telemetry DaemonSets were running. *Gap*: idle time since the previous test ended.

Test	D (s)	T	Fans	Clients	Gap (m:s)
1	124	on	on	0	–
2	124	on	on	0	06:55
3	124	on	on	1	05:37
4	124	on	on	4	05:52
5	124	on	on	8	06:13
6.1	124	off	on	0	09:33
6.2	121	off	on	0	04:15
6.3	139	off	on	0	04:00
7.1	133	on	off	0	11:20
7.2	136	on	off	0	06:11
7.3	139	on	off	0	04:05

4.4.1 Collection overhead

The collection duration was included in every telemetry report (Section 3.3.5). Table 4.11 summarizes the results for dynamic reports, which run once per second and are the only report type relevant here. The aggregate mean was 1.158 ms and the median 0.242 ms, with per-test means ranging from 0.808 ms (Test 5) to 1.527 ms (Test 3). The error-flag bitmask remained at zero on all nodes throughout all runs, confirming that no metric failed to collect.

Table 4.11:
Telemetry collection duration for dynamic reports. Each row summarizes all one-second collection cycles recorded across all four worker nodes during that test. The final row aggregates across all telemetry-enabled tests.

Test	Mean (ms)	Median (ms)
1	0.886	0.248
2	1.285	0.245
3	1.527	0.251
4	1.123	0.247
5	0.808	0.256
7.1	1.043	0.226
7.2	1.304	0.234
7.3	1.290	0.241
All	1.158	0.242

The roughly five-fold gap between the mean and median indicates a right-skewed distribution: the majority of collection cycles complete in approximately 0.24 ms, but occasional outliers pull the mean upward. The median is therefore the better measure of typical collection cost, and it remained stable across all tests. No systematic trend was observed between the number of connected dashboard clients and the collection cost. This is expected, since workers are architecturally decoupled from the client-facing side: each worker performs collection locally and pushes data to the master over a single gRPC stream, with no awareness of how many clients are connected. All distribution to clients is handled entirely by the master.

Comparing wall-clock durations in Table 4.10, Tests 1–5 (telemetry on, 0–8 clients) all completed in 124 s (rounded to whole seconds), while Tests 6.1–6.3 (telemetry off) completed in 124, 121, and 139 s. The 18 s spread within the telemetry-off group exceeds any difference between the two groups, suggesting that overhead, if present at all, is smaller than run-to-run variation. Notably, the longest run (Test 6.3, 139 s) also had the shortest cooling gap since the previous test (4:00), suggesting that residual heat may have contributed to the spread, consistent with the thermal effects observed in the fan-removal tests.

4.4.2 Thermal behavior under load

The Raspberry Pi 5 implements two-stage thermal throttling. When the SoC temperature reaches 80°C, the ARM cores are progressively throttled back; at 85°C, both the ARM cores and GPU are throttled more aggressively [23]. All nodes were enclosed in the official Raspberry Pi 5 Case, which is equipped with a variable-speed fan controlled by the firmware.

Table 4.12 breaks down the time each node spent in four temperature ranges during each test, alongside the mean pod execution time on that node. Figures 4.4 and 4.5 visualize the same data as heatmaps for easier cross-node and cross-test comparison.

In the fan-cooled runs (Test 1-5), nodes 0, 1 and 3 often spent a majority of the workload time in the 80-85°C soft-throttling range, with occasional incursions above 85°C (up to 4.3% for node 0 in test 5). Node 2, by contrast, spent 100% of the workload time below 80°C in every fan-cooled run. Table 4.12 shows that node 2

recorded the shortest mean pod execution time at 104 s (Test 2 and 3).

Removing the fans (tests 7.1-7.3) confirmed the relationship between temperature and performance. Without active cooling, all four nodes spent substantially more time in the 85-90°C hard-throttling range, with the proportion increasing across successive runs as residual heat accumulated between tests.

Despite the clear thermal impact on execution time, the CPU-frequency metric showed 2400 MHz at almost every one-second sample across all telemetry-enabled tests (with only 6 exceptions out of all recorded samples). This may reflect multiple factors. The one-second polling frequency is too coarse to capture transient drops, as the firmware can reduce and restore clock speed within sub-second intervals. Additionally, the kernel interface used to read frequency may not fully reflect firmware-level throttling, in which case the reported value could remain unchanged regardless of polling rate. The gathered metrics can confirm that thermal conditions degraded performance (through temperature data and execution times) but cannot observe the throttling events directly through the frequency metric.

Table 4.12: Percentage of workload time spent in each temperature range per node, alongside the mean pod execution time. Temperature ranges correspond to the Pi 5's throttling thresholds: below 80 °C (no throttling), 80–85 °C (soft throttling), and above 85 °C (hard throttling).

Test	Node	<80°C	80-85 °C	85-90 °C	>90°C	Mean Pod Exec (s)
1	0	43.1%	55.2%	1.7%	0%	117
1	1	37.4%	62.6%	0%	0%	107
1	2	100.0%	0%	0%	0%	114
1	3	26.5%	69.9%	3.5%	0%	112
2	0	40.3%	58.0%	1.7%	0%	120
2	1	52.7%	47.3%	0%	0%	112
2	2	100.0%	0%	0%	0%	104
2	3	22.5%	76.6%	0.9%	0%	109
3	0	38.5%	59.0%	2.6%	0%	118
3	1	39.2%	60.8%	0%	0%	106
3	2	100.0%	0%	0%	0%	104
3	3	28.3%	69.8%	1.9%	0%	110
4	0	38.3%	61.7%	0.0%	0%	120
4	1	33.3%	66.7%	0%	0%	107
4	2	100.0%	0%	0%	0%	117
4	3	21.2%	77.0%	1.8%	0%	110
5	0	31.9%	63.8%	4.3%	0%	115
5	1	48.2%	50.9%	0.9%	0%	113
5	2	100.0%	0%	0%	0%	107
5	3	15.7%	80.6%	3.7%	0%	107
7.1	0	15.7%	26.8%	57.5%	0%	125
7.1	1	17.5%	30.8%	51.7%	0%	119
7.1	2	27.9%	35.1%	36.9%	0%	116
7.1	3	16.4%	23.8%	59.8%	0%	124
7.2	0	10.9%	18.8%	69.5%	0.8%	130
7.2	1	11.3%	22.6%	66.1%	0%	127
7.2	2	17.7%	28.3%	54.0%	0%	116
7.2	3	7.9%	14.3%	74.6%	3.2%	125
7.3	0	5.2%	17.8%	76.3%	0.7%	133
7.3	1	5.7%	19.7%	70.5%	4.1%	127
7.3	2	11.1%	23.9%	65.0%	0%	119
7.3	3	3.2%	9.6%	68.0%	19.2%	131

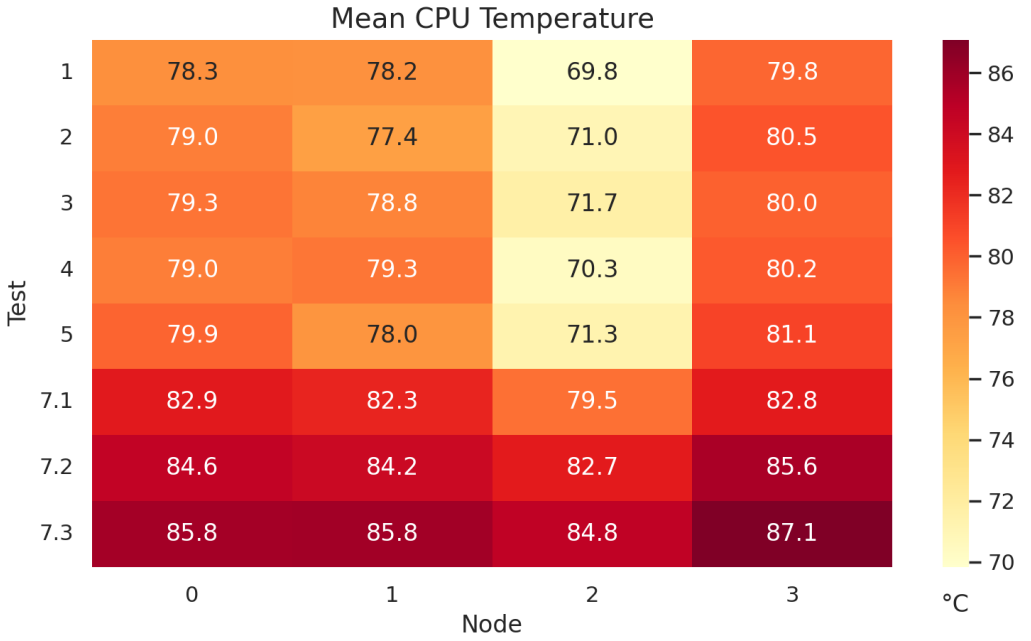


Figure 4.4: Heatmap of mean CPU temperature for each test and node. Node 2 is visibly cooler than the other three nodes in every fan-cooled test (1–5). The fan-removal tests (7.1–7.3) show progressively higher temperatures across all nodes.

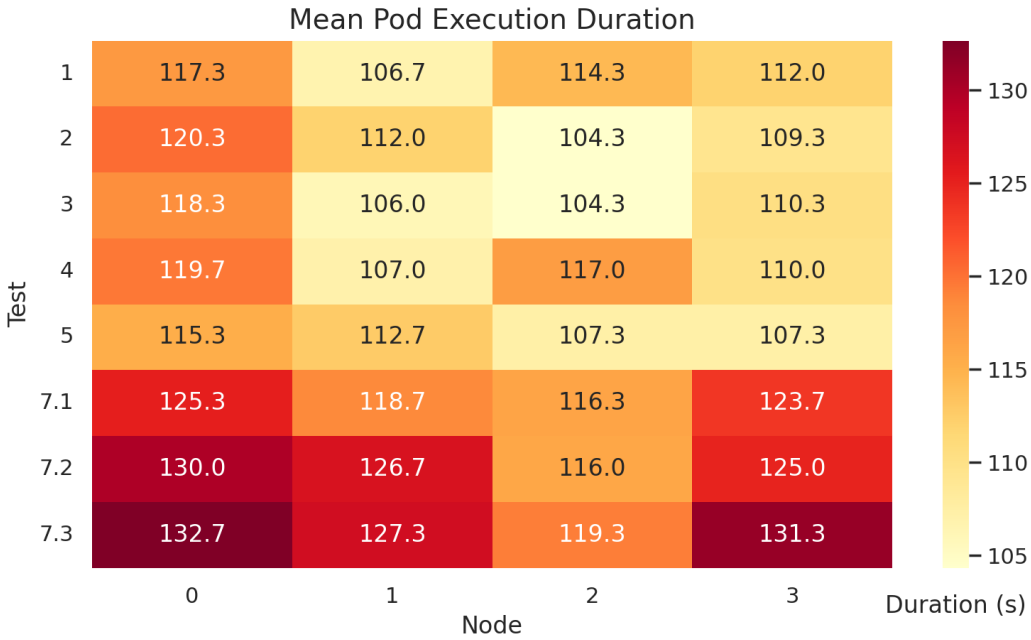


Figure 4.5: Heatmap of mean pod execution duration for each test and node. Longer execution times correlate with higher temperatures, particularly in the fan-removal tests. Node 2 achieves the shortest or near-shortest execution times in most tests, particularly in the fan-removal runs.

4.4.3 Telemetry-driven discovery of hardware heterogeneity

The thermal analysis led to an unplanned finding about the cluster's hardware composition. As shown in Table 4.12 and Figures 4.4 and 4.5, node 2 behaved as a consistent outlier across all eight telemetry-enabled test runs: it was the only node to remain entirely below 80°C during every fan-cooled test, and it achieved the shortest mean pod execution time in most runs. The remaining three nodes showed broadly similar thermal profiles to one another, with all three regularly entering the 80–85°C soft-throttling range under the same workload.

This discrepancy was not expected. All units were acquired together, assembled in identical official cases, and ran the same OS and workload configuration. Because the telemetry data made the anomaly immediately visible — a column of 100% in the <80°C bin where every other node showed a majority in the 80–85°C bin—it prompted a physical inspection of the hardware. The investigation revealed that node 2 reported revision code `d04171` (rev 1.1), while the remaining nodes reported `d04170` (rev 1.0) [24]. Raspberry Pi has not published a detailed changelog between the two revisions, but the Rev 1.1 node consistently exhibited lower operating temperatures and shorter pod execution times across most test runs.

This discovery illustrates the value of continuous per-node monitoring: the discrepancy was invisible during procurement and would not have surfaced from aggregate metrics alone. The performance difference is modest—node 2 completed pods typically 3–13% faster than the slowest node, with the gap widening in the fan-removal runs. The scaling results presented earlier remain valid in aggregate, but per-node comparisons should be interpreted with the revision difference in mind.

4.5 Dashboard

The dashboard was implemented with four functional tabs, shown in Appendix A.1. The MAIN tab provides a cluster-wide overview, reporting the number of online and offline nodes alongside aggregate CPU, memory and disk usage (Figures A.1 and A.2). The NODES tab presents per-node detail, with each card covering the metrics collected by the observability system shown in Table 3.2 (Figures A.3 and A.6). The LOG tab exposes the historical telemetry records for download in JSONL or CSV format (Figure A.8). The GRAPH tab plots a rolling window of live telemetry as time-series, with each node assigned a fixed color across all plots for easy tracking (Figures A.9, A.10 and A.11). Two additional tabs, WORKLOADS and TERMINAL, remain partial implementations discussed in Section 5.3.

5. Discussion

This chapter discusses the results in relation to the project goals, focusing on how the cluster performed across the evaluated workloads, with particular attention to scalability, energy efficiency, system bottlenecks and observability. The chapter also discusses limitations in the evaluation and identifies possible directions for future work.

5.1 Interpretation of results

Storage IO had been identified as a possible bottleneck, particularly given that worker nodes were equipped with SD cards, and the design deliberately mitigated this. The NFS share was hosted on an NVMe SSD on the master (Section 3.1.1), and the telemetry pipeline was built to avoid writing to worker storage entirely. With that mitigation in place, neither storage nor network throughput constrained the workloads in practice. A consistent pattern across the results is that performance was governed primarily by the compute capacity of the individual nodes, with throughput rising and falling with available CPU. This is visible in the matrix multiplication and hashcat results (Section 4.1.1 and 4.2.1), where adding nodes scaled performance near-linearly but adding further pods to already saturated nodes did not, and in the inter-node measurements, where even slow data transfers proved to be processing bound rather than limited by network bandwidth (Section 4.3.2).

The energy results show a trade-off between energy usage and runtime. Configurations with fewer nodes had lower power draw during both the hashcat and matrix multiplication executions (Tables 4.5 and 4.6, Figure 4.3), since fewer Raspberry Pi nodes were actively performing work. However, these configurations also required substantially longer execution times to complete the workload. When more nodes were used, both the power draw and the speedup increased, but the power increase slightly outpaced the runtime reduction, so the shorter runtime only partly offset the higher draw (Table 4.5). The faster configurations were therefore not the most energy efficient. For the pods-per-node tests, the poorer performance of more pods per node suggests that improvement in utilization could not compensate for the increase in overhead and resource contention (Table 4.6).

The same trade-off appears at the level of clock frequency (Figure 4.3). Among the lower clock speeds, reducing the frequency saved energy at the cost of a longer runtime, with no single frequency being best on both counts. An interesting observation is that clock speeds at 2000 MHz and above consumed more energy than 1900 MHz without completing the workload any faster, and the 2400 MHz drew the most energy of any configuration while running marginally slower than 1900 MHz. Pushing the clock beyond roughly 1900 MHz therefore yielded no benefits, possibly because thermal throttling at the highest clocks prevented the additional power from translating into additional throughput. As with the diminishing returns

from adding pods beyond two per node, the fastest-clocked configuration was not the most efficient, and in this case not even the fastest.

The degree and nature of scaling varied between workloads. Matrix multiplication and hashcat both scaled near-linearly as nodes were added, reaching speedups of 3.79x and 3.63x on four nodes respectively (Tables 4.1 and 4.3), effectively the same scaling behavior. LLM inference produced different results: while distributing inference across nodes did not reduce execution time (Table 4.7), it enabled models to run that would not fit within the RAM constraints of a single SBC (Section 4.3.2). This allowed the cluster to execute larger and more capable models, at the cost of increased latency. Since the distribution was handled by llama.cpp rather than implemented as part of this project, the absence of a speedup may reflect the particular tool, its configuration, or other factors not investigated here, rather than a fundamental limit. The results therefore suggest that the SBC cluster was effective for workloads that can be cleanly parallelized, and that it provided a means of running models too large for a single node, though whether such distribution can be made performant on this hardware remains open.

The decision to build a custom observability system rather than adopt an existing solution shaped much of what the telemetry pipeline could and could not do. The motivation was concrete, pull-based tools assume that each node can run an HTTP server and buffer metrics to local storage (Section 2.6.1), neither of which suits storage constrained workers running from SD cards. The push based design followed directly from this constraint, and it was what allowed the system to collect metrics without imposing any writes on worker storage. The cost of this choice was that monitoring infrastructure had to be built and maintained in-house rather than drawn from a mature ecosystem, but in return the system remained small, transparent and fully under the project's control.

The telemetry system was most clearly validated especially by its usefulness, along with its overhead figures. Continuous per-node monitoring surfaced a hardware revision difference between otherwise identical nodes that was invisible at procurement (Section 4.4.3) and the real time dashboard made problems such as misconfigured nodes, uneven load and thermal behavior visible as they happened. At the same time, the evaluation exposed a genuine limitation of the system: while it could establish that thermal throttling was degrading performance through temperature and execution time data, the CPU frequency metric reported the maximum clock at almost every sample and could not capture the throttling events (Section 4.4.2). The system could therefore confirm that throttling occurred without observing it directly, a reminder that the usefulness of a telemetry pipeline depends not only on its overhead but also on whether the metrics it samples can capture the phenomena of interest.

5.2 Limitations and sources of errors

Due to time constraints, the scope of benchmarking performed in this project was limited. Although the matrix multiplication workloads executed 10 internal itera-

tions per pod, each test configuration was only submitted once. As a result, both wall-clock time and pod execution time represent single measurements. The remaining workloads were also executed only a limited number of times, and most benchmarks were relatively short in duration. It is possible that effects such as thermal throttling could have had a greater impact during longer-running workloads.

Another potential source of uncertainty stems from Kubernetes scheduling behavior. Since pod placement and resource allocation are handled dynamically by the Kubernetes scheduler, slight variations in node assignment or resource contention may have influenced execution times between tests. These factors were not explicitly controlled for during testing.

As discussed in Section 4.4.3, hardware heterogeneity within the cluster was discovered during the course of the project. This introduced uncertainty into the results of the benchmarks. Reproducing the tests could thus yield different outcomes, depending on hardware composition.

The conclusions regarding scalability should also be interpreted with caution. The cluster consisted of only four worker nodes, which limits the extent to which scaling behavior could be evaluated. Thus, it remains unclear at what scale diminishing returns or significant overheads would begin to appear, since testing did not extend beyond the available cluster size.

Finally, the matrix multiplication relied on Python and NumPy. While NumPy provides highly optimized numerical operations, Python itself is an interpreted language and introduces some level of runtime overhead compared to lower-level compiled languages such as C or C++.

5.3 Future directions

Several extensions could strengthen the system. Incorporating live telemetry into the job scheduling could allow workload distribution to account for current node conditions such as temperature and load, rather than relying solely on a static hardware survey. Expanding the cluster with more nodes and intentionally heterogeneous hardware could test scaling beyond the current setup. Re-implementing compute-intensive workloads in a compiled language could reduce interpreter overhead and provide a more accurate measure of the cluster's raw computational throughput.

On the infrastructure side, replacing Raspberry Pi OS with a purpose-built Kubernetes operating system such as Talos could reduce the base resource footprint on each node and simplify provisioning. PXE boot could further streamline scaling by allowing new nodes to boot directly from the network, replacing the golden image approach that proved error-prone during the project. Integrating LLM inference into the dashboard could give users a chat interface backed by the cluster's pooled memory. Two dashboard features also remain incomplete: a drag-and-drop workload submission interface and an interactive per-node shell, both of which could reduce the need for direct SSH access to the cluster.

6. Conclusion

This chapter answers the four research questions posed in Section 1.2, concerning energy efficiency, scalability, inter-node communication and telemetry overhead, and answers each in turn.

Energy efficiency. Under active load the clusters energy efficiency is characterized by a trade-off between power draw and runtime. Configurations with more nodes drew more power but completed workloads faster, and the faster configurations were not the most energy efficient. The same held at the clock-frequency level, where clock speeds above roughly 1900 MHz cost more energy with no reduction in runtime. Among the tested configurations, the four-node setup with two pods per node offered the best balance. Idle power draw was measured to 14 W, rising to 24 W under full four node LLM inference (master node excluded in both numbers). The research question also concerned long term efficiency under intermittent, low-utilization conditions. This could not be measured directly, as the power meter used reports only instantaneous power draw rather than accumulated energy over time. All energy figures in this report are therefore derived from short benchmark runs. Characterizing long term efficiency for irregular workloads remains an open question for future work, and would require either an energy integrating meter or continuous logging of instantaneous draw over realistic duty cycles.

Scalability. Computational throughput scaled close to linearly with the number of nodes: matrix multiplication reached a speedup of 3.79x, and hashcat 3.63x on the same number of nodes. Scaling by increasing pods per node instead yielded diminishing returns beyond two pods per node, attributable to CPU contention and scheduling overhead rather than to the workload itself. Expanding the cluster was reasonably practical, though not fully automated. Each new node required OS provisioning and an Ansible playbook run to install K3s, after which it joined the cluster and the observability DaemonSets began reporting without further per-node configuration. This only needs to be performed once, and given that in practice a node remains in a cluster for months to years, the provisioning cost is negligible. However, with only four worker nodes available, the scale at which diminishing returns or coordination overhead would begin to dominate could not be determined.

Inter-node communication. Network and storage throughput did not emerge as the primary bottleneck for the workloads tested. The NFS share removed the need for significant local storage on the worker nodes, and transfer times remained small for smaller files. The dominant limitation was instead CPU performance: during distributed LLM inference, the large model was loaded into RAM across the worker nodes as slow as roughly 30 MiB/s on a cold load, well below the gigabit network's capacity of around 120 MiB/s, indicating that processing overhead from reading and distributing the model (rather than network bandwidth) constrained the load time.

Observability. The custom telemetry system met its central goal of monitoring the

cluster in real time without writing any metric data to the storage-limited worker nodes. The cost of collecting each dynamic report was consistently low, with a median of 0.242 ms across all telemetry-enabled tests, a figure backed by the thousands of individual one-second readings taken across the nodes. Its effect on workload completion time is harder to pin down: although each submission ran ten internal iterations, the comparison rests on the wall-clock duration of a single submission per configuration, and the 18 s spread between identically configured telemetry-off runs exceeds any difference between the telemetry-on and telemetry-off groups. This is therefore best read as evidence that any overhead is smaller than the measurement uncertainty, rather than as proof of zero cost.

References

[1] C. Ramon-Cortes, P. Alvarez, F. Lordan, J. Alvarez, J. Ejarque, and R. M. Badia, "A survey on the distributed computing stack," *Computer Science Review*, vol. 42, p. 100 422, 2021, ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2021.100422. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013721000629.

[2] W. Ahmed and Y. W. Wu, "A survey on reliability in distributed systems," *Journal of Computer and System Sciences*, vol. 79, no. 8, pp. 1243–1255, 2013, ISSN: 0022-0000. DOI: https://doi.org/10.1016/j.jcss.2013.02.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000013000652.

[3] R. Gohil and H. Patel, "Comparative analysis of cloud platform: Amazon web service, microsoft azure, and google cloud provider: A review," in *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, 2024, pp. 1–5. DOI: 10.1109/ICCCNT61001.2024.10725914.

[4] P. Kayal, "Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope : Invited paper," in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, 2020, pp. 1–6. DOI: 10.1109/WF-IoT48130.2020.9221340.

[5] C. Severance, "Eben upton: Raspberry pi," *Computer*, vol. 46, no. 10, pp. 14–16, 2013. DOI: 10.1109/MC.2013.349.

[6] H. Penyala, S. Ibrahim, and A. El Mesalami, "The raspberry pi education mine: For teaching engineering and computer science students concepts like, computer clusters, parallel computing, and distributed computing," in *2020 IEEE International Conference on Electro Information Technology (EIT)*, 2020, pp. 624–628. DOI: 10.1109/EIT48999.2020.9208242.

[7] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A container-based edge cloud paas architecture based on raspberry pi clusters," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, 2016, pp. 117–124. DOI: 10.1109/W-FiCloud.2016.36.

[8] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, pp. 65–77, 2019, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.12.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918302507.

[9] 6sense, *Best containerization software in 2026*, https://6sense.com/tech/containerization, Accessed: 2026-05-13, 2026.

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016, ISSN: 0001-0782. DOI: 10.1145/2890784. [Online]. Available: https://doi.org/10.1145/2890784.

[11] A. Salvi, Y. Shinde, P. Munde, S. Mhadgut, and B. Masram, "Synergizing infrastructure as code and container orchestration: A survey on terraform and kubernetes automation," *International Journal of Engineering Research & Technology (IJERT)*, vol. 15, no. 01, Jan. 2026. DOI: 10.17577/IJERTV15IS010223. [Online]. Available: https://www.ijert.org/research/synergizing-infrastructure-as-code-and-container-orchestration-a-survey-on-terraform-and-kubernetes--IJERTV15IS010223.pdf.

[12] K3s, *K3s - lightweight kubernetes*, https://docs.k3s.io, Accessed: 2026-05-14.

[13] K. Pijanowski and M. Galarnyk, *What is distributed training?* https://www.anyscale.com/blog/what-is-distributed-training?source=techstories.org, Accessed: 2026-05-31, Apr. 2022.

[14] Hashcat, *Hashcat*, https://github.com/hashcat/hashcat/tree/master, [Accessed 5-May-2026], 2026.

[15] J. Lee, S. Park, J. Kwon, J. Oh, and Y. Kwon, *Exploring the trade-offs: Quantization methods, task difficulty, and model size in large language models from edge to giant*, 2025. arXiv: 2409.11055 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2409.11055.

[16] Prometheus, *Overview - prometheus documentation*, https://prometheus.io/docs/introduction/overview, Accessed: 2026-05-14.

[17] gRPC, *Grpc - faq*, https://grpc.io/docs/what-is-grpc/faq, Accessed: 2026-05-14.

[18] Google, *Protocol buffers documentation*, https://protobuf.dev, Accessed: 2026-05-14.

[19] N. Gupta et al., "Deploying a task-based runtime system on raspberry pi clusters," in *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2020, pp. 11–20. DOI: 10.1109/ESPM251964.2020.00007.

[20] Y. Lee and D. Park, "Exploring the effects and potential of unlocked i/o-powered single board computer clusters," *Scientific Reports*, vol. 16, no. 1, p. 4486, 2026. DOI: 10.1038/s41598-025-34623-x. [Online]. Available: https://www.nature.com/articles/s41598-025-34623-x.

[21] B. Qureshi and A. Koubaa, "On energy efficiency and performance evaluation of single board computer based clusters: A hadoop case study," *Electronics*, vol. 8, no. 2, p. 182, 2019. DOI: 10.3390/electronics8020182. [Online]. Available: https://www.mdpi.com/2079-9292/8/2/182.

[22] Z. Li, T. Li, W. Feng, M. Guizani, and H. Yu, *Prima.cpp: Speeding up 70b-scale llm inference on low-resource everyday home clusters*, 2025. arXiv: 2504.08791 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2504.08791.

[23] A. Allan, *Heating and cooling raspberry pi 5*, https://www.raspberrypi.com/news/heating-and-cooling-raspberry-pi-5/, Accessed: 2026-05-14, Raspberry Pi, Oct. 2023.

[24] Raspberry Pi, *Raspberry pi revision codes*, https://www.raspberrypi.com/documentation/computers/raspberry-pi.html#raspberry-pi-revision-codes, Accessed: 2026-05-14, Raspberry Pi.

A. Figures

A.1 Dashboard screenshots

The following figures accompany the discussion of the dashboard in Section 4.5.

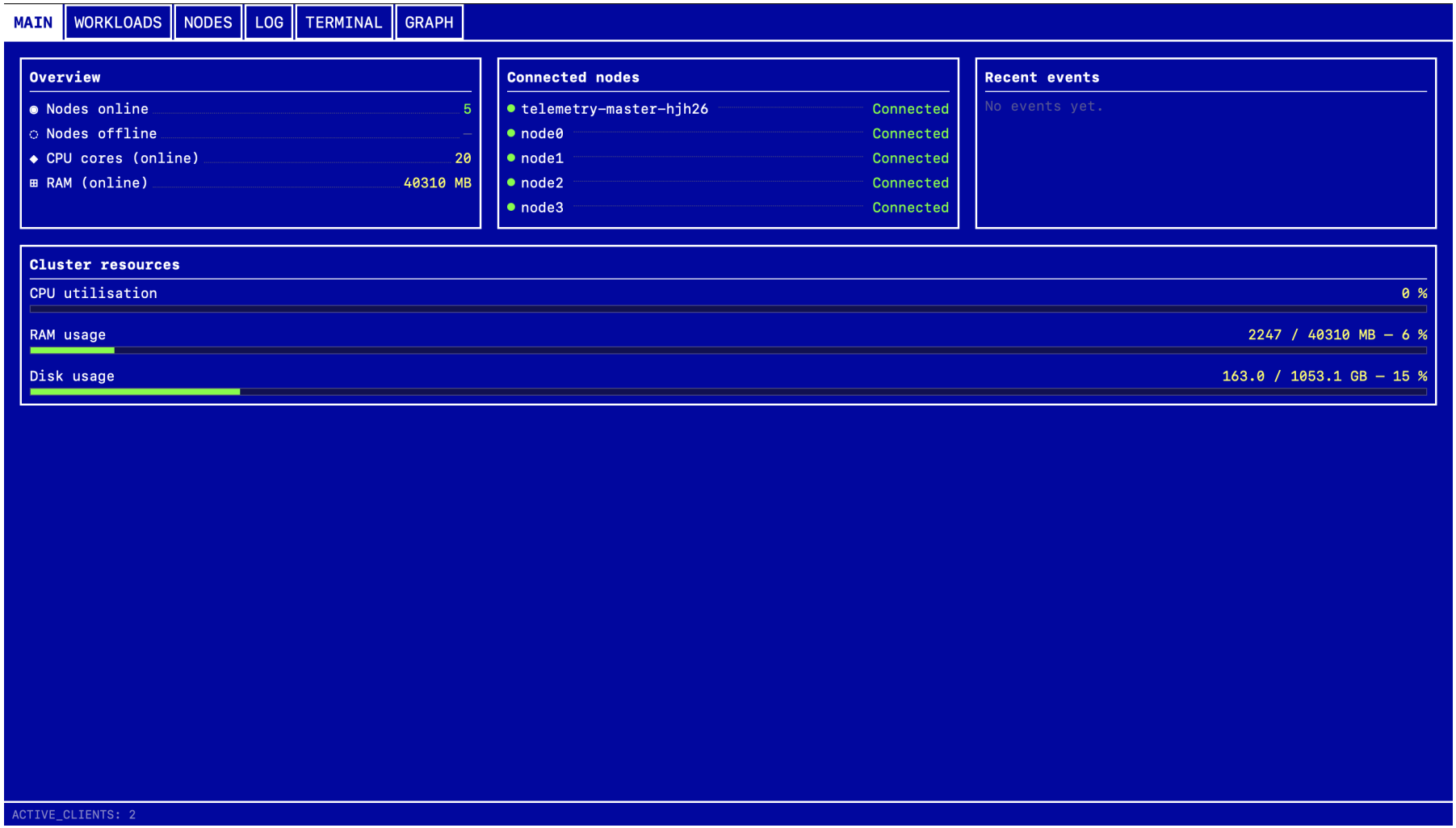


Figure A.1: The MAIN tab with all five nodes connected. Aggregate resource figures reflect the full cluster.

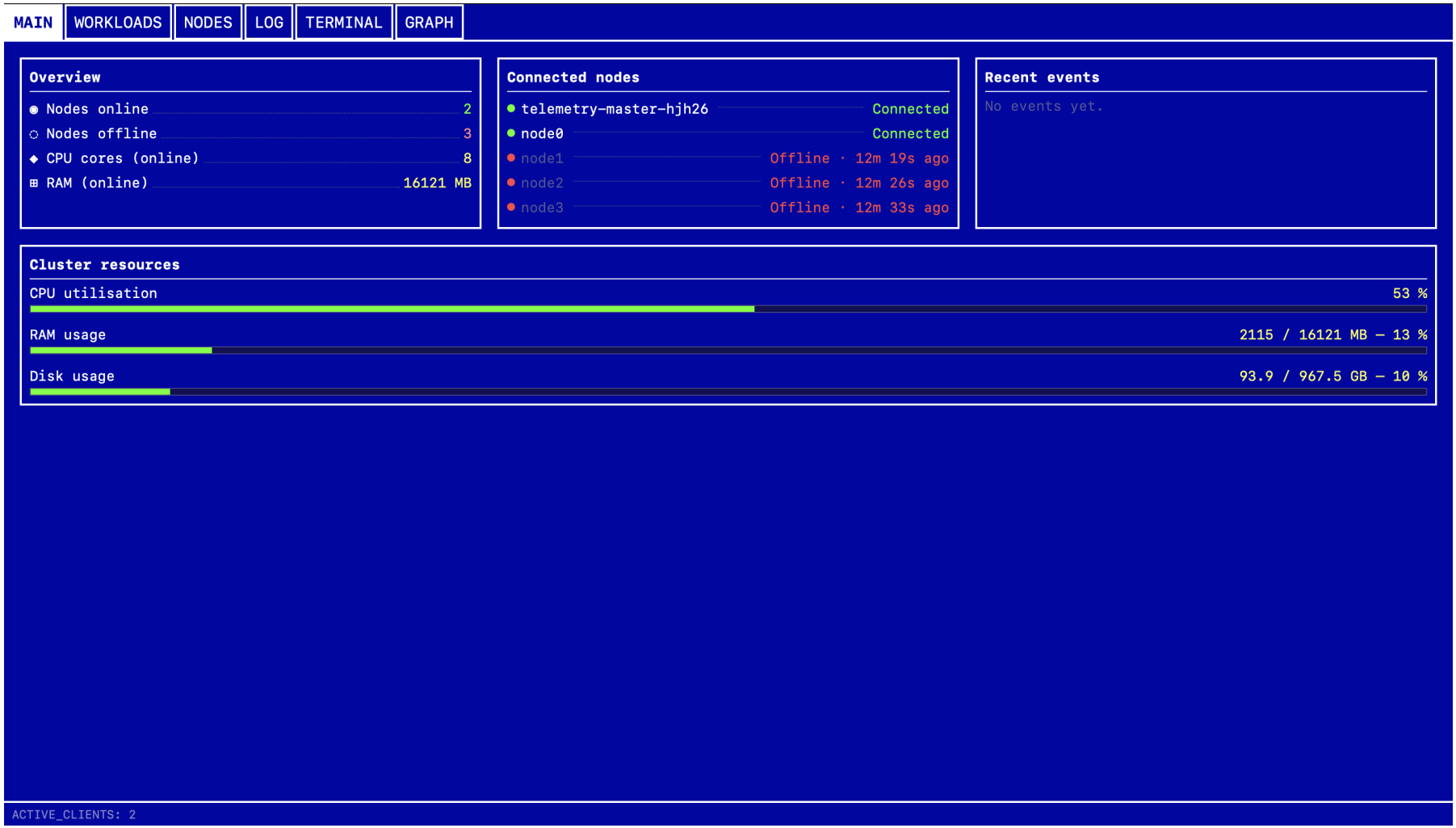


Figure A.2: The MAIN tab after three workers have gone offline. Offline nodes remain visible in the Connected nodes panel, annotated with the time since the last received report.

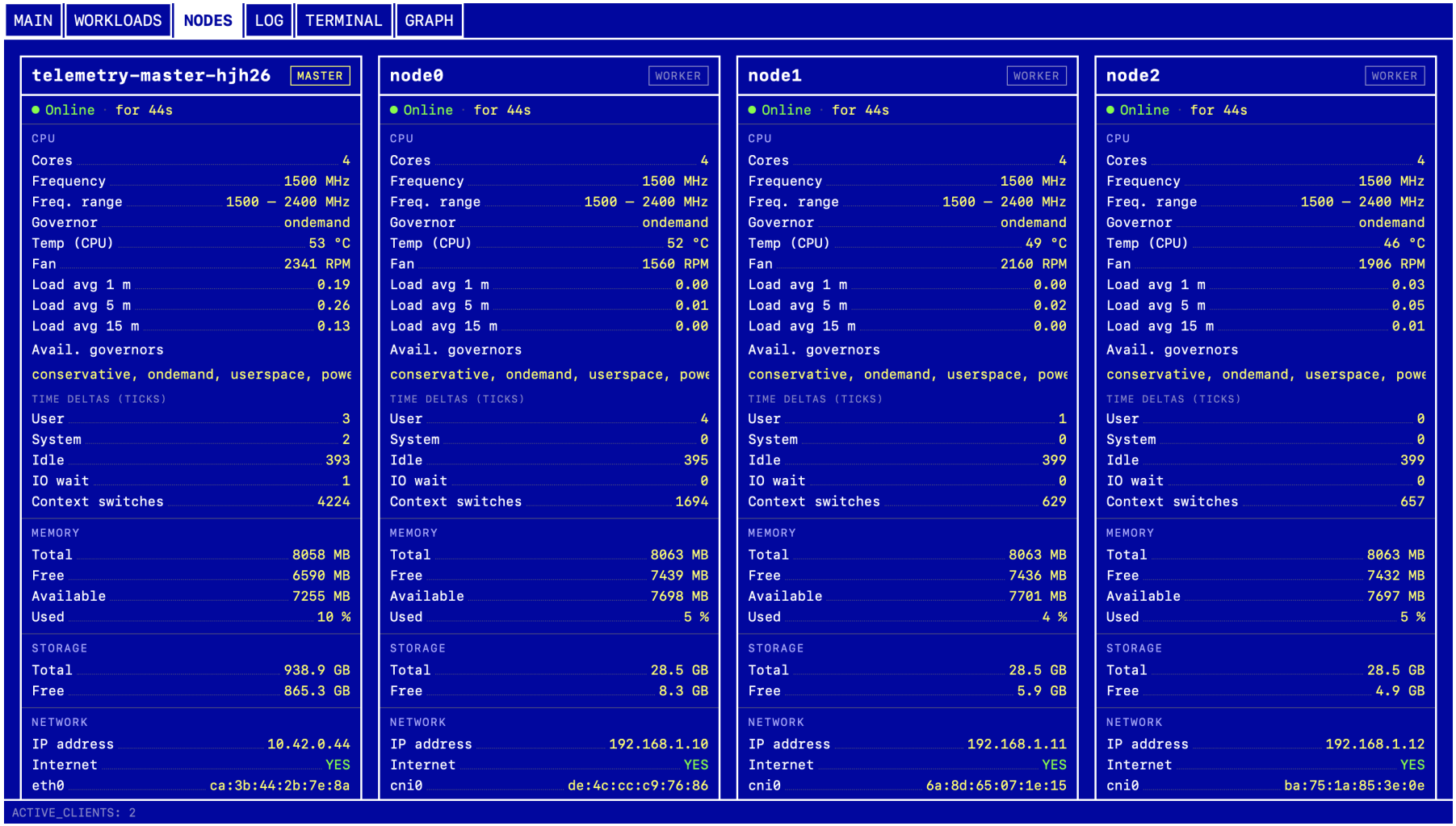


Figure A.3: The NODES tab with all nodes connected. The master node is distinguished from workers by a MASTER badge and by its hostname being highlighted.

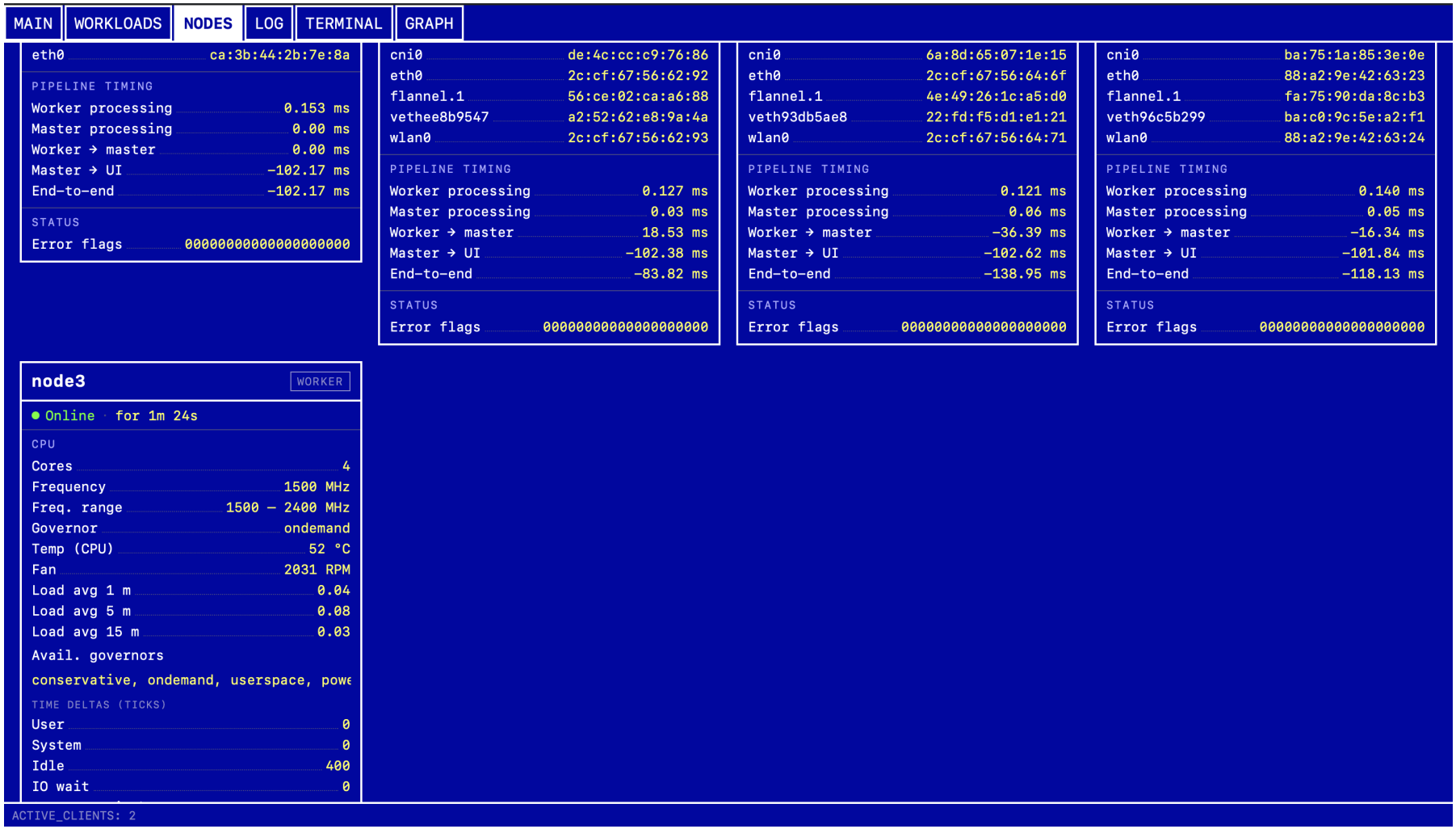


Figure A.4: Continuation of NODES page

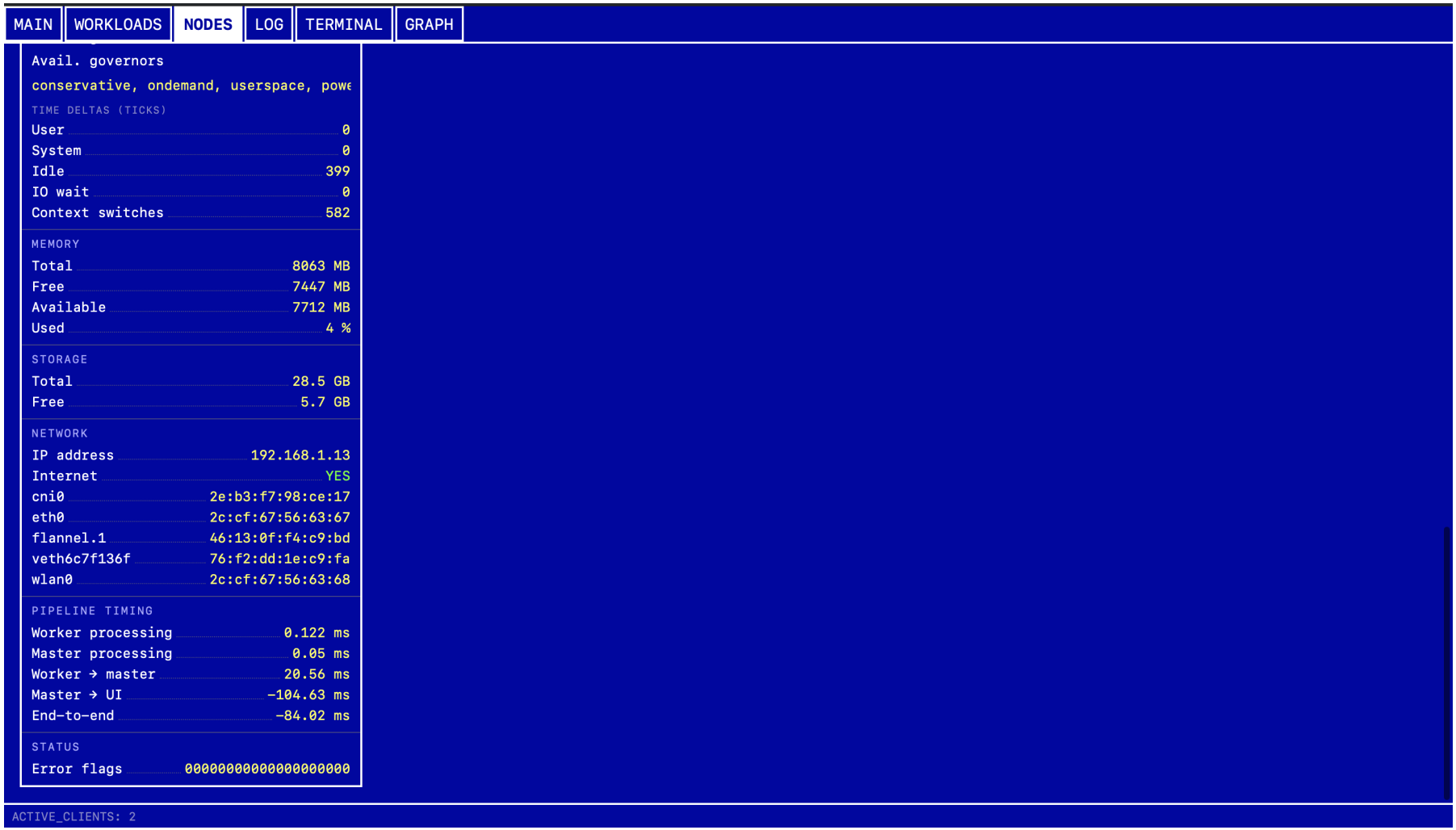


Figure A.5: Bottom section of NODES page

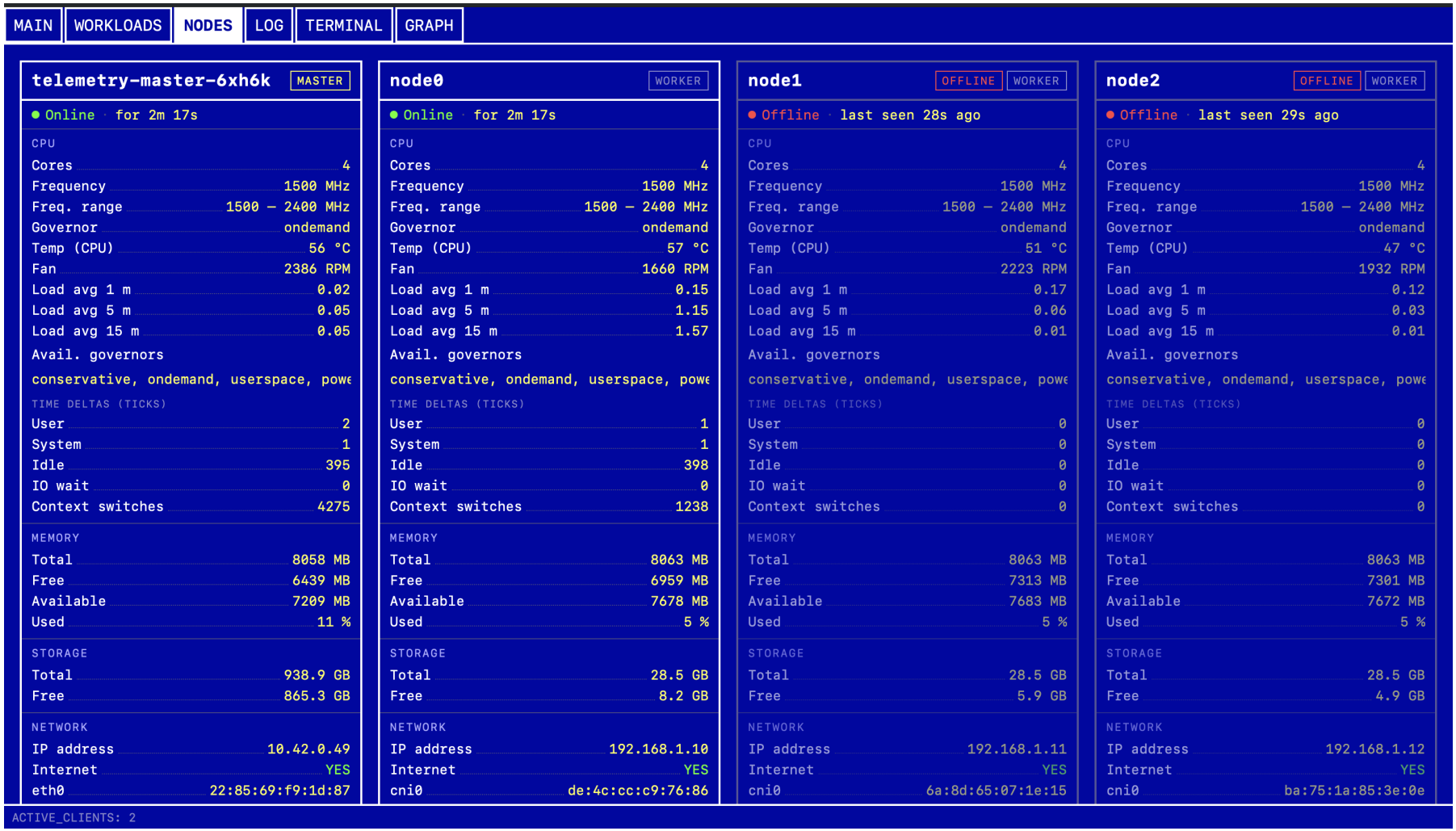


Figure A.6: The NODES tab with two workers offline. Offline nodes retain their last reported values and are annotated with an OFFLINE badge and a last-seen timestamp.

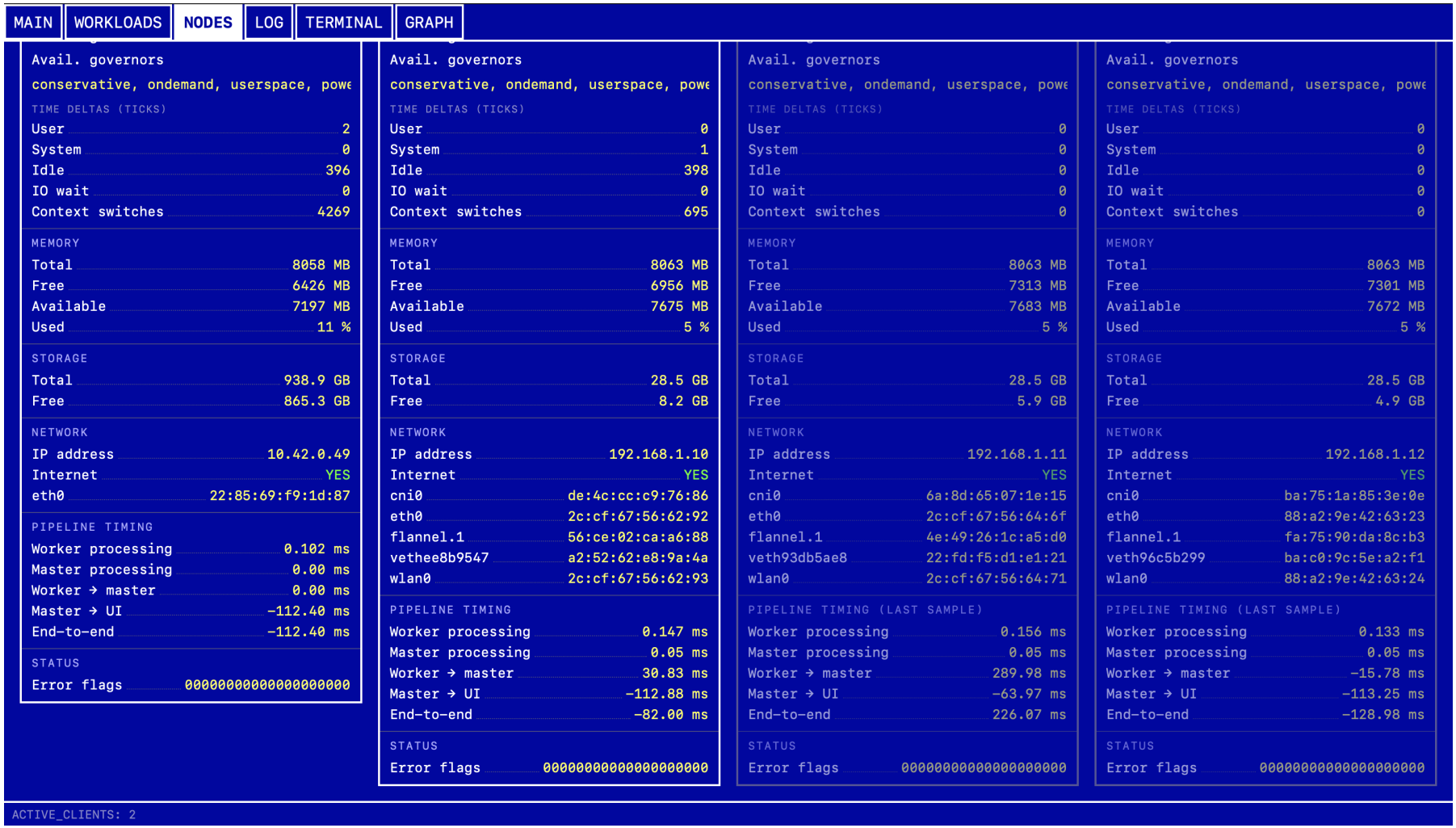


Figure A.7: Bottom section of the NODES tab with some nodes offline

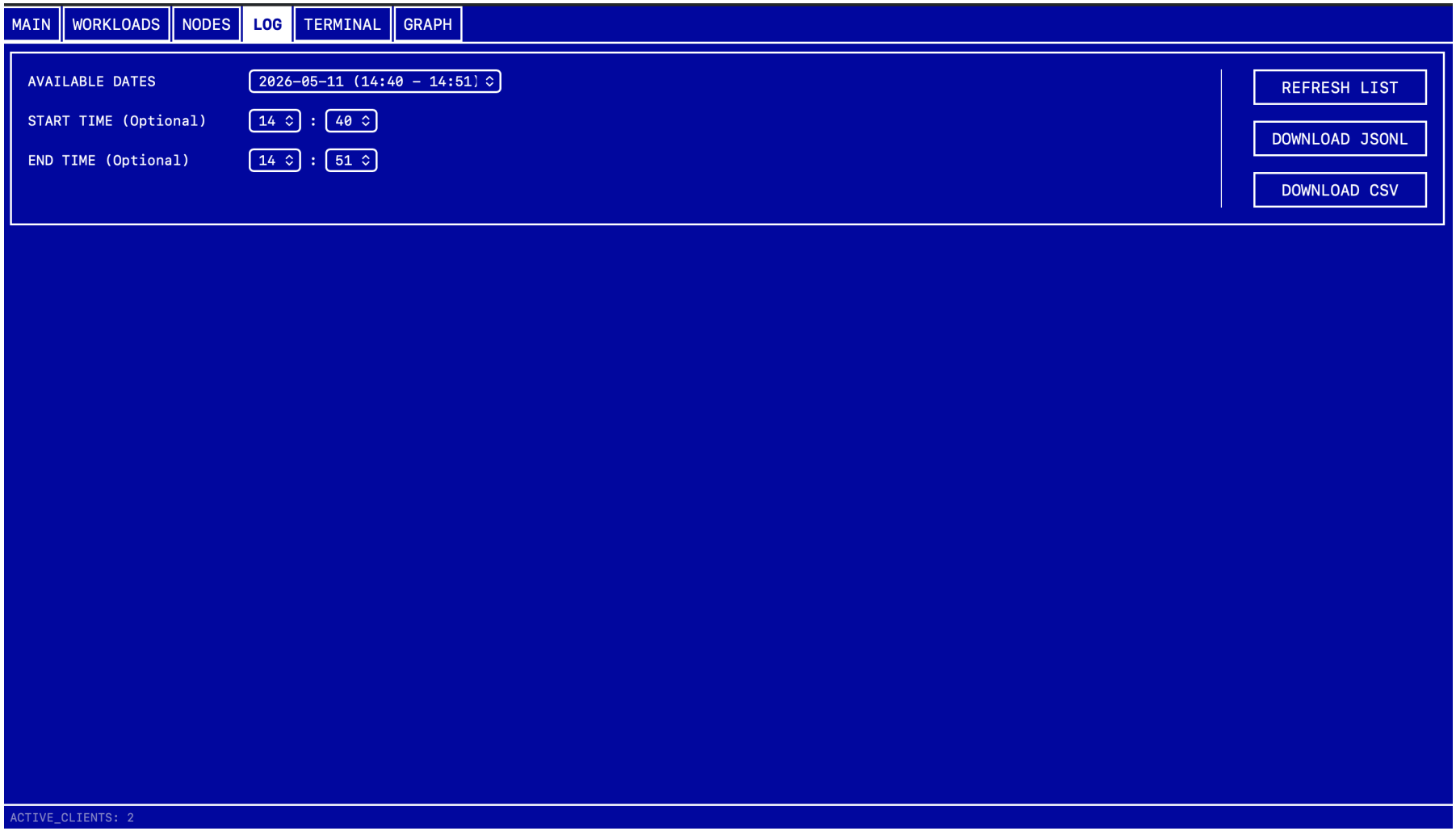


Figure A.8: The LOG tab, providing controls for retrieving stored telemetry over a chosen time range and downloading it in either JSONL or CSV format.

Figure A.9: The GRAPH tab, showing CPU temperature, current frequency, and the user and system time counters as live time-series across the cluster.

Figure A.10: The GRAPH tab, showing CPU idle time, I/O wait, context switches and fan RPM metrics.

Figure A.11: The GRAPH tab, showing free and available memory on each node together with the worker and master processing latencies introduced by the telemetry pipeline.

Figure A.12: The WORKLOADS tab in its current placeholder state. The intended interaction allows a user to drop a workload file directly into the browser, select a workload type, and monitor execution through the queue and result panels.

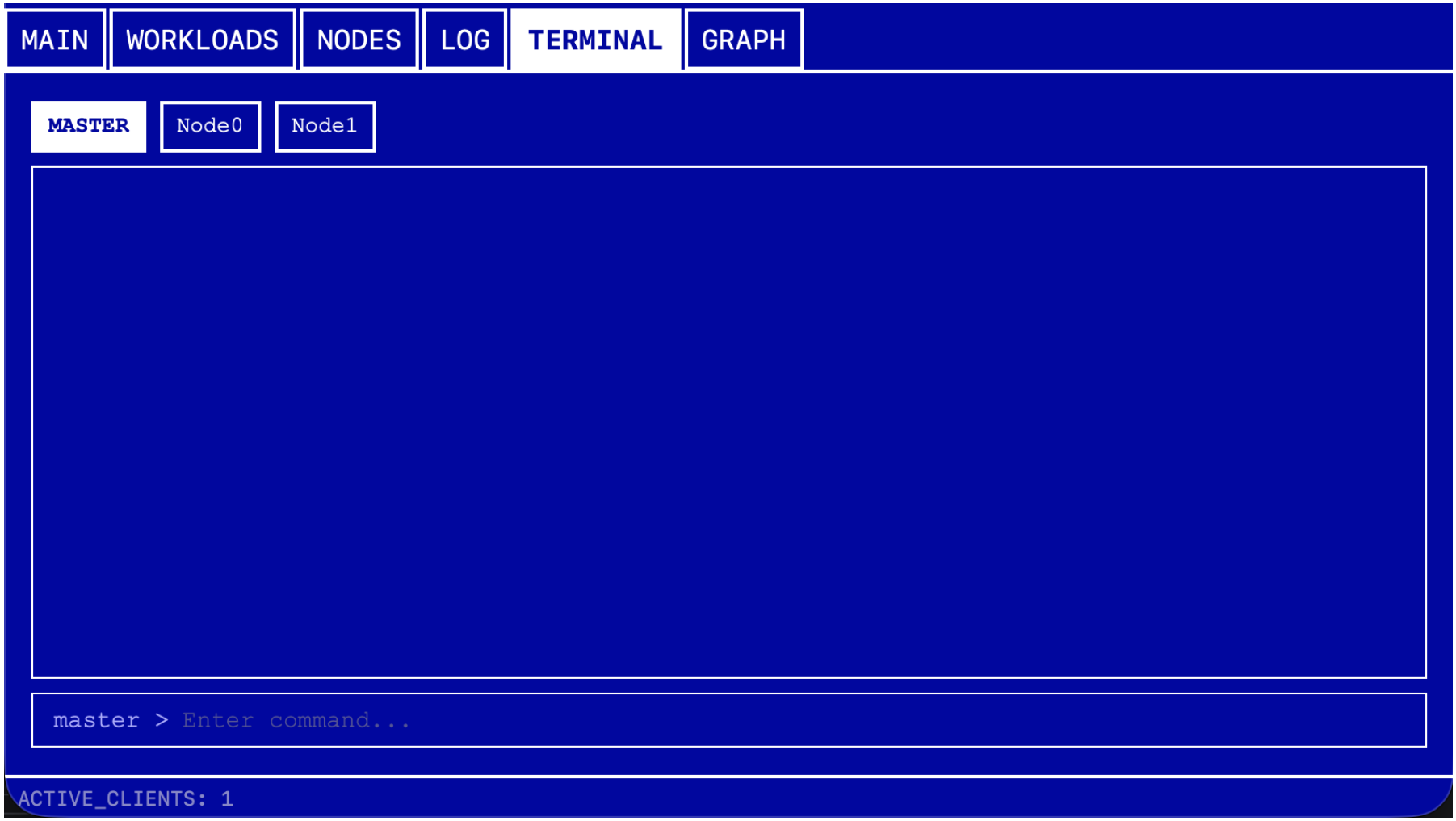


Figure A.13: The TERMINAL tab in its current placeholder state. The full implementation vision would allow users to use this window as an SSH replacement with direct access to each node's own command-line tool