GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Master's Thesis

submitted in partial fulfillment of the
requirements for the course "Applied Data Science"

Dynamically Negotiating Resources Between Kubernetes and Slurm Environments

---

Shrinath Madde

MatrNr: 25235677

First Supervisor: Dr. Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

ISSN: 1612-6793

September 26, 2026

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

Abstract

Research computing centers increasingly host two kinds of workload on the same Graphics Processing Unit (GPU) hardware: classic Slurm batch jobs, and long-running services such as Large Language Model (LLM) inference that are a natural fit for Kubernetes. Today a site keeps the two apart by drawing a fixed line through the hardware, assigning some GPU nodes to each cluster. This split wastes hardware, because the two workloads follow different daily rhythms: GWDG traces analyzed here show inference demand swinging almost sixtyfold between its daily peak and trough, while the Slurm GPU pool stays nearly flat and backlogged around the clock, so nodes reserved for inference sit idle overnight while the batch queue is still full. Existing systems either host the service inside Slurm, inheriting batch-queue latency and rebuilding what Kubernetes already offers, or place Kubernetes inside a Slurm allocation, forfeiting bare-metal performance; none move bare-metal GPU nodes between a real Kubernetes inference cluster and a Slurm cluster on a shared stateless provisioning base, driven by live demand.

This thesis proposes and evaluates the *negotiation service*: a single daemon on the Warewulf provisioning host that observes both clusters and moves GPU nodes across the boundary automatically by reassigning a node's boot image and rebooting it, with nothing from the previous role surviving. The service carries each node through a five-phase transition that drains running work before it moves, so no batch job is killed and no inference request is dropped. The thesis proposes two decision policies: an on-demand rule that reacts to current load, and a predictive policy that forecasts the inference demand of the coming hour with a pretrained, zero-shot time-series model and provisions to the upper band of that forecast.

The evaluation replays a real six-day Slurm `sacct` trace (26,667 GPU jobs) and a real month-long inference trace (about 936,000 requests) from GWDG's production systems, lined up on the same calendar days, through a discrete-event simulator. Sharing a dynamic pool of about an eighth of the fleet cut the mean Slurm queue wait from 290 minutes under a static split to 166 minutes under the predictive policy (a 43% drop, and 83% at the median), while fewer than one inference request in 160 had to wait for capacity. Replacing the forecast with the true future load, which no real system can know, shortened the wait by only one further minute, so an ordinary forecast already collects nearly all of the queue benefit. The results show that the fixed line between an inference cluster and a batch cluster is not required: A small control loop on the provisioning host can shift it with the daily rhythm of the workloads at little cost to the inference service.

I hereby declare that all uses of tools and AI as well as assistance from third parties in this work have been fully and accurately disclosed in the Transparency Report in Appendix Section A.
Missing or incorrect information may be considered an attempt to cheat.

Acknowledgements

I would like to thank my supervisor, Dr. Jonathan Decker, for his continuous support and feedback over almost a year. He taught me how to approach and think about a research problem, how to draft it, and, most importantly, how to present it. His skill and enthusiasm in research, scientific writing, and teaching have been truly inspiring.

I would also like to thank Prof. Dr. Julian Kunkel for always sharing his wisdom and experience, both about work and about life. I learned something new in every meeting with him, and each one gave me a new perspective. I have always admired his problem-solving skills and dedication.

Contents

List of Tables **v**

List of Figures **vi**

List of Abbreviations **vii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Goals and research questions . . . 1
1.3 Contributions . . . 2
1.4 Outline . . . 3

2 Background **4**
2.1 HPC architecture and workloads . . . 4
2.2 Warewulf . . . 7
2.3 The Slurm Workload Manager . . . 10
2.4 Kubernetes . . . 16

3 Related Work **25**
3.1 The Balancer . . . 25
3.2 The Torque-Operator . . . 26
3.3 High-Performance Kubernetes . . . 26
3.4 Kind Slurm Integration . . . 27
3.5 Ephemeral Kubernetes . . . 28
3.6 SAIA: Slurm-native hosting for HPC-based services . . . 35
3.7 Inference scaling with a Kubernetes front end . . . 35
3.8 Workload forecasting and Chronos-Bolt . . . 36
3.9 Summary and positioning . . . 38

4 Methodology **40**
4.1 Research questions as design problems . . . 40
4.2 Design of the negotiation service . . . 40
4.3 Metrics . . . 54
4.4 Experiments . . . 55

5 Evaluation **60**
5.1 Transition cost . . . 60
5.2 Workload characterization . . . 61
5.3 Policy definitions . . . 65
5.4 Forecast accuracy . . . 66
5.5 Queue wait, inference service level, and idle capacity . . . 68
5.6 Transitions over one day . . . 71
5.7 Sensitivity analysis . . . 72
5.8 Answers to the research questions . . . 74

6 Discussion **77**
6.1 Limitations 77
6.2 Implications for computing centers 78
6.3 Beyond Slurm, Kubernetes, and LLMs 78
6.4 Open problems 79
6.5 Research gaps 79

7 Conclusion **81**
7.1 Summary of contributions 81
7.2 Future work 81

References **83**

A Transparency Report: Tools and Third-Party Contributions **A1**
A.1 Overview A1
A.2 Per Section A1

List of Tables

1 Related systems against the four requirements 39
2 Daily workload shape on both sides 49
3 Files of the reference implementation 53
4 Metrics collected in the evaluation 55
5 Simulation setup 57
6 Prototype testbed 58
7 Software and code revisions 59
8 Measured time for one node transition 60
9 Estimated GPU footprint of the inference services 64
10 Forecast accuracy of Chronos-Bolt 68
11 Main simulation results 69
12 Idle capacity per policy 70

List of Figures

1 Reference High-Performance Computing (HPC) cluster architecture 5
2 Warewulf boot workflow 9
3 Slurm daemon architecture 11
4 Kubernetes daemon architecture 17
5 Admin-kubeconfig distribution in Ephemeral Kubernetes 20
6 Ephemeral Kubernetes control leader and NFS share components 30
7 Ephemeral Kubernetes deployment sequence 32
8 Overlay layout for the two roles 43
9 Example `node-config.conf` 44
10 Architecture of the negotiation service 45
11 One transition in each direction 47
12 Static baseline split 48
13 The predictive policy on real data 51
14 Inference arrivals over one month 62
15 Inference daily and weekly pattern 62
16 Slurm GPU occupancy by hour of day 64
17 Negotiation results 69
18 When the nodes move during a day 71
19 Sensitivity to movable-pool size 73
20 Where in the forecast range to provision 73
21 Sensitivity to the safety floor 74
22 Sensitivity to move time 75

List of Abbreviations

API Application Programming Interface

CNI Container Network Interface

CPU Central Processing Unit

CRI Container Runtime Interface

GPU Graphics Processing Unit

HPC High-Performance Computing

IPVS IP Virtual Server

KV Key–Value

LLM Large Language Model

PKI Public Key Infrastructure

QOS Quality of Service

1 Introduction

1.1 Motivation

GWDG, the scientific computing center in Göttingen, operates two kinds of workload that both depend on the same expensive GPU nodes. The first is classic HPC: Researchers submit batch jobs, and the Slurm workload manager [YJG03] queues them and dispatches them to the GPU nodes of the Grete cluster. The second is LLM inference: The Chat AI service [Doo+25] answers requests from thousands of users, and every answer is computed on the same kind of A100 GPU that the batch jobs occupy. Today the hardware is divided by a fixed line: Some nodes belong to the batch cluster, others serve inference, and a node never changes sides.

The two workloads follow different daily rhythms, as two traces from GWDG's production systems show (Section 5.2). Inference traffic follows the working day. It climbs through the morning, peaks around midday, and falls close to zero at night; averaged over the month, the busiest hour of the day carries almost sixty times the load of the quietest, and weekends are quieter still. Slurm's GPU demand follows no such rhythm. Batch jobs occupy the cluster around the clock, the daytime load lies only four percent above the overnight load, and the queue never empties: The median GPU job waits about two hours before it starts. So every night, nodes reserved for inference sit idle while batch jobs wait for the same kind of node.

The fixed line has two costs. The first is idle hardware. The inference side must own enough nodes for its midday peak, so most of that capacity sits idle through the night and the weekend, and none of it can be handed to the waiting batch jobs. The second is duplicated machinery. A long-running service fits a batch scheduler poorly, so GWDG's inference platform SAIA hosts its LLM services through Slurm itself and has to rebuild what Kubernetes provides by default: service discovery, load balancing, health checks, and automatic restarts [Doo+25]. Moving the services onto a real Kubernetes cluster would remove that duplicated work, but under a fixed split it would also lock yet another set of nodes into a role that is only busy half the day.

Nothing in the hardware enforces this line. GWDG's compute nodes are stateless: They keep no operating system on a local disk and boot over the network from an image served by Warewulf. A node's role is therefore set by the image it boots. After a new image assignment and a reboot, a Slurm compute node comes back as a Kubernetes worker a few minutes later, with nothing of its old role left behind. Ephemeral Kubernetes [DK25] has already shown that a complete Kubernetes cluster can be provisioned this way on HPC hardware. What is still missing is a component that watches the demand on both sides and moves nodes automatically to the side that needs them. This thesis builds that component, the *negotiation service*: a single daemon on the Warewulf host that lends idle inference nodes to Slurm and takes them back before the inference load returns.

1.2 Goals and research questions

The goal of this thesis is to design, build, and evaluate the negotiation service. The work asks one central question:

> Can a single Warewulf-managed cluster serve both Kubernetes-based service workloads and Slurm batch jobs on the same physical nodes, with those nodes

moved between the two systems automatically in response to demand?

That question breaks down into four research questions.

- **RQ1: Can a Warewulf-provisioned cluster switch the same physical node between a Kubernetes role and a Slurm role?** The same node must boot as a Kubernetes worker one time and as a Slurm compute node the next, with the role set only by its image assignment on the Warewulf host. The other three questions depend on this one.
- **RQ2: Can the negotiation service switch a node's role automatically, without disrupting running work?** A switch must need no manual steps, and it must not kill a running batch job or drop an inference request. The service therefore has to drain a node, wait for its work to finish, and only then move it.
- **RQ3: Does automatic node sharing reduce idle node time and Slurm queue wait compared to a static split, and does predicting the inference load help beyond reacting to it?** Moving nodes costs time on both sides, so it pays off only if the Slurm queue gets noticeably shorter and fewer nodes sit idle. The thesis compares an on-demand policy that reacts to the current load with a predictive policy that forecasts the inference demand of the coming hour, using a pretrained time-series model [Ans+24], and moves nodes before the change arrives.
- **RQ4: What is the cost of one transition, and which cooldown period stops the system from oscillating?** A node in transition serves neither side, so the cost of a single move decides how often moving pays off. The system must also not shuttle the same node back and forth faster than either side can benefit from it.

1.3 Contributions

A working prototype and a trace-driven simulation answer the research questions. The thesis presents four contributions.

- **A negotiation service that moves nodes between Kubernetes and Slurm.** The service is a single daemon on the Warewulf host, about 1,600 lines of Bash around tools the site already operates, and it installs no new software on any compute node. It carries a node through five phases (drain, reassign, reboot, join, verify) in both directions with no human step in between. This answers RQ1 and RQ2.
- **Two policies for deciding when a node moves.** The on-demand policy lends a node the moment Slurm has a job waiting and no node free. The predictive policy forecasts the inference load of the coming hour with a pretrained time series model [Ans+24] whose weights are downloaded once and never trained on GWDG data, and moves nodes before the load changes. This answers the policy half of RQ3.
- **A measurement of what one role switch costs.** On a GWDG testbed the prototype required a median of 98 seconds from Kubernetes to Slurm and 102 seconds for the return. A node that has just moved must then stay put for five minutes, about two and a half times the two minutes the simulation assumes per move, and this rule alone kept every policy from moving the same node back and forth. This answers RQ4.

- **An evaluation on two production traces from the same days.** A discrete-event simulator replays the inference requests of one month of Chat AI traffic (about 936 thousand) together with the 26,667 A100 GPU jobs of a six-day Slurm accounting export. Lending idle inference nodes to Slurm cut the median batch queue wait by 83% and the mean by 43%, while fewer than one inference request in 160 was delayed, and giving the policy the true future load instead of a forecast, which no real system can know, shortened the queue by only one further minute. This answers the rest of RQ3.

The workload study behind that evaluation is also useful on its own. Two traces recorded on the same days show that inference demand swings almost sixtyfold over the day while batch GPU demand stays flat and backlogged. The negotiation service pays off only where this pattern exists, so a site can check its own traces for it before building anything.

1.4 Outline

The rest of the thesis is organized as follows. Section 2 explains the building blocks: HPC clusters and their workloads, Warewulf, Slurm, and Kubernetes. Section 3 reviews earlier systems that combine batch and service workloads, covers the two pieces of earlier work this thesis builds on directly (Ephemeral Kubernetes and the Chronos-Bolt forecasting model), and shows the gap this thesis fills. Section 4 turns the research questions into design problems, presents the negotiation service itself (the node role model, the transition protocol, and the two decision policies), and defines the metrics and experiments. Section 5 reports the results and their limits. Section 6 interprets those results beyond GWDG, and Section 7 summarizes the contributions and the work that remains.

2 Background

This chapter presents the technologies the work depends on, in the order in which they fit together. It begins with how an HPC cluster is built and which workloads it serves (Section 2.1). It then describes Warewulf *(Section 2.2), which installs software on compute nodes without keeping any state on them;* Slurm *(Section 2.3), the scheduler that hands out nodes to HPC batch jobs; and* Kubernetes *(Section 2.4), the system that operates long-lived services in containers. Two further pieces the negotiation service builds on directly, the Ephemeral Kubernetes deployment and the Chronos-Bolt forecasting model, are earlier work of their own, so they are covered in Section 3.5 and Section 3.8 together with the rest of the related work. The purpose here is only to explain how each piece works; comparisons with other approaches follow in Section 3.*

2.1 HPC architecture and workloads

A modern HPC cluster is a group of Linux servers wired together and operated as one shared system by many users at once. The exact size and the local rules differ from site to site, but the same basic layout (sketched in Figure 1) recurs at most academic and national centers, GWDG included. The rest of this chapter assumes that layout: Warewulf provisions the compute nodes, Slurm decides which user receives which nodes, Kubernetes hosts service workloads on the same hardware, and the negotiation service shifts nodes between the two.

The rest of this subsection presents how a cluster is assembled (Section 2.1.1), how trust is handled when many users share it (Section 2.1.2), the workloads it has to serve (Section 2.1.3), how compute nodes are kept stateless (Section 2.1.4), and the recent move toward hosting both batch jobs and long-running services on the same hardware (Section 2.1.5), which motivates the rest of the thesis.

2.1.1 Cluster anatomy

A cluster has four kinds of nodes connected by two networks, as shown in Figure 1.

Login (head) nodes - Login nodes are the only machines users can SSH into directly, and a site usually provides only a few of them. Users log in here to edit code, compile it, prepare input data, and submit jobs. They are not intended for heavy computation; executing a long simulation on a login node counts as misuse.

Compute nodes - Most of the hardware lives in compute nodes, which hold the CPUs and GPUs that execute user jobs. Users cannot reach these nodes directly; jobs arrive on them only through the workload manager. A typical site provides several flavors of compute node: plain CPU-only nodes for ordinary simulations, GPU nodes for machine learning and other accelerated work, and big-memory nodes for jobs that need more RAM than a standard server provides. The nodes that the negotiation service moves between Slurm and Kubernetes are the GPU ones.

Management nodes - A few machines host the cluster's control plane: the workload manager's controller, the accounting database, the provisioning system that delivers operating-system images to compute nodes, monitoring tools, and the entry points for external user interfaces. The Warewulf host, the Slurm controller, the Kubernetes control plane, and, in this thesis, the negotiation service all run on management nodes.

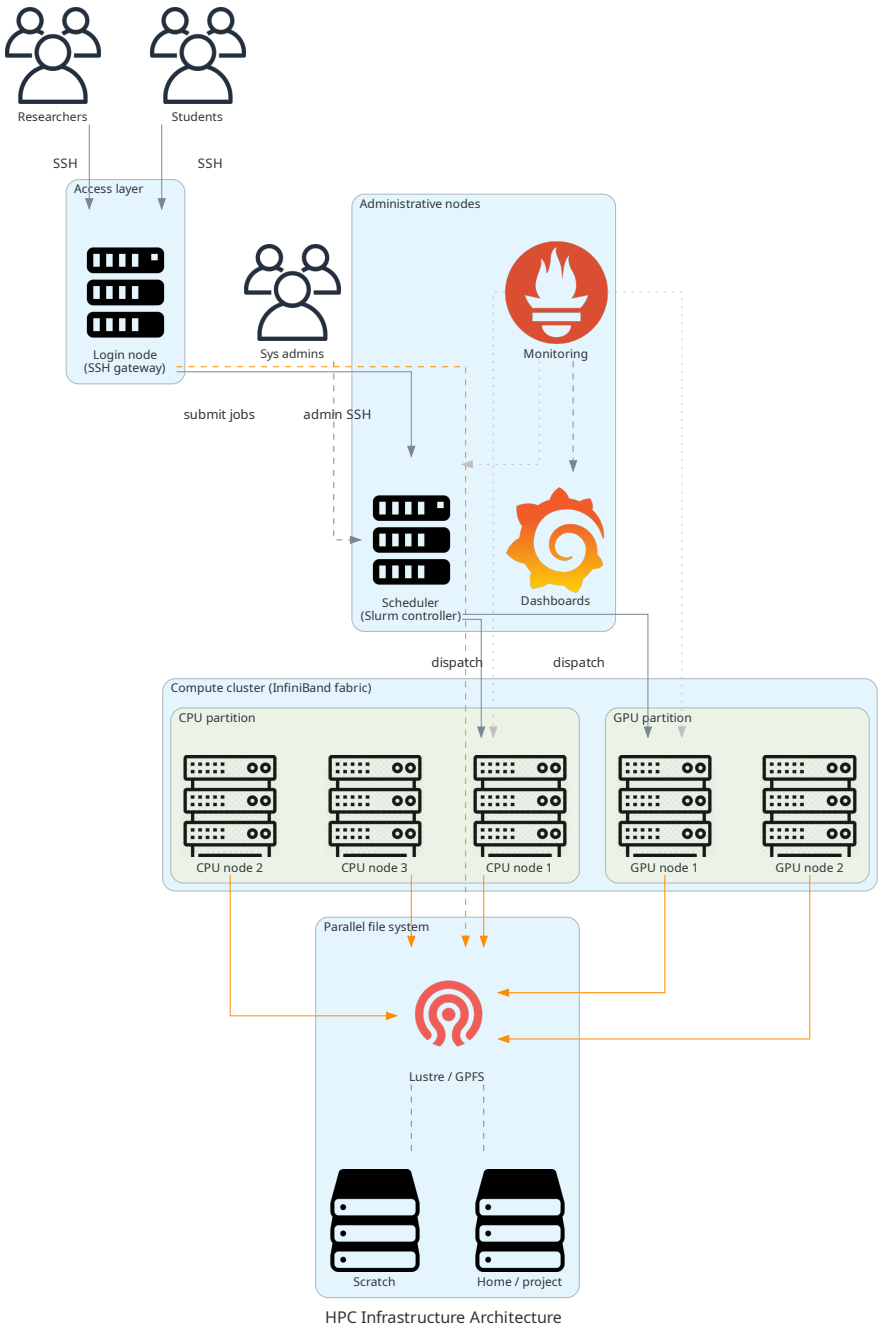


Figure 1: Reference HPC cluster layout: Users SSH into a login node, which submits jobs to the Slurm controller; the controller dispatches them to CPU and GPU compute nodes on an InfiniBand fabric, and every node mounts the same parallel file system. Generated with the `diagrams` Python library; source at `scripts/Hpc_architecture.py`.

Storage - User data is kept on a shared parallel filesystem (Lustre, BeeGFS, GPFS, or similar) that every node mounts at the same path. This filesystem is the only place where data is meant to survive long-term: Compute nodes hold nothing important on their own and can be swapped out at any time.

Two networks - Each compute node sits on two separate networks. The first is a high-speed interconnect (InfiniBand, Slingshot, or RoCE) that carries the traffic between job processes, where a few extra microseconds can slow a job down. The second is a slower Ethernet management network, which provisions nodes, collects logs and metrics, and carries workload-manager commands. Separating the two prevents job traffic from blocking control traffic and prevents management traffic from delaying jobs.

2.1.2 The multi-tenant trust model

HPC clusters are shared by many users at once. Dozens or hundreds of researchers from different groups share the same hardware, and much of their data on the parallel filesystem is sensitive or unpublished. Three consequences of this shared model recur throughout the thesis.

No root access for users - Users receive shell access only on login nodes, and they may never become root anywhere. Anything they need that the system does not provide reaches them through user-space tools such as Lmod, Spack, or EasyBuild, or through rootless containers such as Apptainer or Singularity. This is also why a plain Kubernetes deployment on an HPC cluster is difficult: Kubernetes assumes it owns root on every node it manages, which is the privilege HPC sites withhold. That conflict shapes the Ephemeral Kubernetes design (Section 3.5).

Resource accounting - Compute time is treated as a budget. Every job is billed to an account, fair-share priorities prevent heavy users from blocking everyone else, and usage reports return to funding agencies. Because of that, moving a node out of Slurm and into Kubernetes has to stop its billing cleanly: While a node is acting as a Kubernetes worker, no Slurm user should be charged for it.

Strict separation of concerns - A cluster functions only if workloads stay clear of one another, and each layer of the stack enforces that separately. The workload manager isolates jobs on a node with cgroups and namespaces, so one user's job cannot consume another's CPU or memory. The split between the high-speed interconnect and the management network keeps user traffic from interfering with cluster commands. The parallel filesystem relies on POSIX permissions and group membership to keep sensitive data readable only by the people who own it. Any new component added to the cluster, including the negotiation service and the inference platform of this thesis, has to meet the same standard.

2.1.3 Workload mix

Three kinds of workload share the hardware of a modern cluster.

Traditional batch jobs - The classic HPC workload is a batch job with a bounded duration: an MPI simulation, a parameter sweep, or a data-processing pipeline. The user writes a script, requests N nodes for T hours, and the workload manager starts it once those nodes free up. The job writes its output to the parallel filesystem, terminates, and the nodes return to the pool. Slurm was designed to schedule this kind of job.

Interactive and on-demand work - Researchers also need interactive sessions for debugging, profiling, and exploratory analysis. Web portals such as Open OnDemand sit in front of the workload manager and translate requests such as "give me a Jupyter notebook on a GPU for two hours" into ordinary batch allocations. To Slurm this work is indistinguishable from any other batch job; only the user-facing entry point differs.

Service workloads - The newest category is the long-running service. Examples include a database that backs a research collaboration, a continuous-integration runner, or, more and more often, an LLM inference endpoint [Kwo+23] that has to answer HTTP requests at all times. These workloads have no completion point; they run until someone stops them. They require a stable network address, automatic restart after a crash, and rolling updates without downtime. Batch managers were never designed to provide that, whereas container orchestrators such as Kubernetes were. The requirement to host both kinds of workload on the same hardware creates the negotiation problem this thesis addresses, and it has motivated recent work on hosting services natively on HPC infrastructure, both through Slurm itself [Doo+25] and through short-lived Kubernetes clusters built on Warewulf [DK25].

2.1.4 Stateless compute nodes

Compute nodes usually keep no state of their own. There is no normal operating system installed on a local disk; instead, each node boots over the network from an image served by a provisioning system, with a small per-node overlay layered on top for configuration. The image is the single source of truth for the software a node holds, so a reboot suffices to deliver a software update or to change the node's role. The negotiation service depends on this property. In a hand-installed, stateful cluster, transferring a node from one workload manager to another would require reinstalling the operating system; with image-based provisioning it amounts to a change of image assignment and a reboot. GWDG and this thesis rely on Warewulf for this, presented next in Section 2.2.

2.1.5 Mixed batch and service clusters

For a long time, a site would buy one cluster for batch work and a separate set of long-running servers for any service it had to host. Three pressures have made that split harder to defend. First, GPU nodes have become so expensive that a site can no longer afford to leave them idle in a single-purpose deployment. Second, the same researchers who submit batch jobs now also expect LLM services from the same site. Third, every separate infrastructure needs its own provisioning, monitoring, and access control, which adds administrative effort. Sites such as GWDG now host both kinds of workload on the same hardware. The question this thesis addresses is how to divide that shared hardware between a batch scheduler and a container orchestrator continuously, according to the demand on each side, rather than partitioning it once and accepting that part of it will sit idle.

2.2 Warewulf

Warewulf[1] is the stateless provisioning system GWDG operates, and the rest of this thesis assumes it. The basic idea was already introduced in Section 2.1.4: Compute nodes

[1]Warewulf: `https://warewulf.org`, accessed 3 April 2026.

have no disk to keep state on, so they boot from images served over the network, with a small per-node overlay layered on top for configuration. This subsection presents the specific Warewulf components that Slurm, Kubernetes, Ephemeral Kubernetes, and the negotiation service all rely on.

Boot cycle - A node managed by Warewulf has no operating system on its local disk; it fetches everything over the network at power-on. The full boot path has five steps, shown in Figure 2. (1) The node's network card sends a DHCP broadcast through its PXE firmware to find out how to boot. (2) The Warewulf server's DHCP service answers with an IP address, a hostname, and the address of the TFTP server that holds the bootloader. (3) The node fetches the small iPXE bootloader over TFTP and executes it; iPXE then opens an HTTP connection to the `warewulfd` daemon on the Warewulf server. (4) `warewulfd` assembles the node's root filesystem on demand: It reads the shared read-only image for the node's role (for example the Slurm image that ships `slurmd`, or the Kubernetes image that ships `kubelet`), lays the per-node overlay on top (hostname, network settings, `slurm.conf` or kubeadm tokens, and so on), and sends the kernel, the initramfs, and the assembled image back to the node, which unpacks them into RAM (or a local cache) and starts `systemd` as PID 1. (5) Once user space is up, the node mounts shared directories (home, project data, software stacks) from a separate NFS server, so user data stays shared across the cluster while the operating system itself lives only in RAM. The operator never touches the node itself: `wwctl` on the server changes the node's image assignment or triggers a reboot. The negotiation service uses these two operations to switch a node between roles.

Images and overlays - A Warewulf *image* is a read-only base operating system shared by a group of nodes. For a Slurm compute role this image holds the kernel, system libraries, and the Slurm `slurmd` daemon; for a Kubernetes worker role in this thesis it holds the Kubernetes `kubelet` together with `containerd`. An *overlay* is a small set of files that Warewulf drops on top of that image to give each node its own identity and configuration.

Warewulf 4 splits overlays into two kinds, depending on when they are applied. A *system overlay* is built into the initramfs and applied once, before `systemd` starts. It carries the files the node needs in order to come up at all: its hostname, its network configuration, its SSH host keys, `/etc/fstab`, and the initial `passwd` and `group` files. Without this overlay every booted node would be identical, with no means of distinguishing them. A *runtime overlay* is fetched later over HTTP by the small `wwclient` agent on the node, and can be refreshed without a reboot. It holds files that may need to change while the cluster is running: the current user and group lists, SSH `authorized_keys`, mount tables, and any per-node configuration that an operator might want to edit after the node has booted. When the operator edits one of these files on the Warewulf host and invokes `wwctl overlay build`, every node receives the new version on its next refresh.

Overlays are also divided by scope. A *per-node* overlay carries items that are unique to one machine (its hostname, its IP address, its BMC settings). A *profile* overlay is attached to a named group of nodes and carries items that are the same for every node of that group (the daemon configuration for the role, shared keys, common mount tables). A node normally receives several overlays at once: a generic shared overlay (`wwinit`) plus one or more role-specific overlays.

Separating the immutable image from the mutable overlay allows many nodes to share one image while each retains its own identity and credentials.

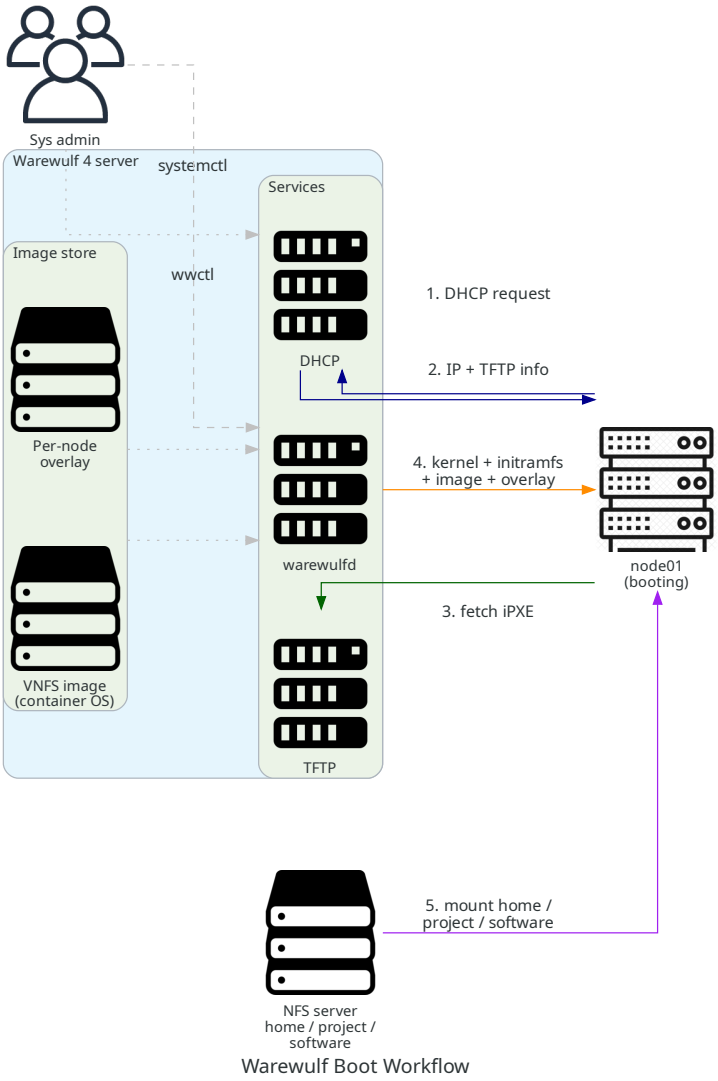


Figure 2: Warewulf 4 boot workflow for a stateless compute node. The five numbered edges (DHCP request; IP and TFTP info; iPXE fetch; kernel, image and overlay delivery; NFS mount of shared data) are walked through step by step in the surrounding text. Generated with the `diagrams` Python library; source at `scripts/Warewulf_workflow.py`.

Operator interface - The `wwctl` command-line tool provides the operations this thesis depends on. `wwctl node set` switches a node to a different image or profile, `wwctl overlay build` rebuilds per-node overlay archives, and `wwctl power cycle` triggers a reboot through IPMI/BMC. Together, these commands let an external program change a node's role with a single reboot, and the negotiation service uses them for every move.

Short-lived clusters on Warewulf - Warewulf has traditionally provisioned long-lived HPC compute nodes, but it can also provision short-lived clusters. Decker and Kunkel [DK25][2] use it to build Ephemeral Kubernetes (Section 3.5), which the negotiation service builds on. The service installs and removes no software on a running node; it only changes the node's image assignment in Warewulf.

[2]Reference implementation and source: `https://github.com/gwdg/pub-2025-ephemeral-kubernetes`, accessed 2 April 2026.

2.3 The Slurm Workload Manager

Slurm [YJG03] (Simple Linux Utility for Resource Management) is the most widely used open-source workload manager for HPC clusters. It started at Lawrence Livermore National Laboratory in the early 2000s and is now maintained by SchedMD. Today it runs on many TOP500 systems and schedules the production workloads at GWDG. Slurm performs three functions at once: It allocates compute resources to users for a bounded period; it provides a framework for launching, monitoring, and terminating parallel work on those resources; and it decides which requests start first when users compete for resources, following a scheduling policy the operator configures. Slurm is licensed under the GNU GPL. SchedMD maintains the full reference documentation[3]; what follows summarizes the parts relevant to this thesis.

The rest of this subsection covers the daemon architecture (Section 2.3.1), the configuration that ties the daemons together (Section 2.3.2), the resource model (Section 2.3.3), job submission (Section 2.3.4), partitions (Section 2.3.5), scheduling (Section 2.3.6), fair-share (Section 2.3.7), node states (Section 2.3.8), power saving (Section 2.3.9), failure handling (Section 2.3.10), and the operator commands (Section 2.3.11).

2.3.1 Daemon architecture

A Slurm cluster consists of a small set of background processes (*daemons*) that communicate over the management network, sketched in Figure 3. One of them is the controller on a management node, one is an agent on every compute node, and a few are helpers for accounting and remote access. Munge[4] authenticates every message they exchange, the shared-secret service covered in Section 2.3.2.

slurmctld: The controller - `slurmctld` is the controlling process of the cluster. It runs on one management node and is the only process permitted to decide scheduling: which job starts next, on which nodes, with which resources. For speed it holds the full state of the cluster (the list of nodes, the queue of jobs, the partitions, and so on) in memory and writes it to disk regularly, so that it recovers after a restart. Every user or operator command (`sbatch`, `srun`, `scontrol`, and so on) is a network call into `slurmctld`. A second controller can be configured as a hot standby that assumes control within seconds if the primary fails.

slurmd: The node agent - `slurmd` runs as root on every compute node and acts as the controller's local agent. At startup it reports the node's hardware (CPUs, memory, GPUs) to `slurmctld` and then waits for instructions. It answers regular health-check pings, starts the user's job when the controller instructs it to, and reports back when the job terminates. It takes no decision of its own. For this thesis, `slurmd` also signals which role a node currently holds: If the daemon is running, the node is a Slurm node; if the node has been re-imaged for Kubernetes, `slurmd` is not running, and Slurm marks the node `DOWN` and schedules nothing on it.

slurmstepd: The per-job helper - Whenever the controller instructs `slurmd` to start a job on its node, `slurmd` spawns a short-lived helper called `slurmstepd` for that one job. `slurmstepd` prepares the job's sandbox (cgroups, environment variables, working

[3]Slurm Workload Manager Documentation: `https://slurm.schedmd.com/documentation.html`, accessed 5 May 2026.

[4]MUNGE Uid 'N' Gid Emporium: `https://dun.github.io/munge/`, accessed 22 April 2026.

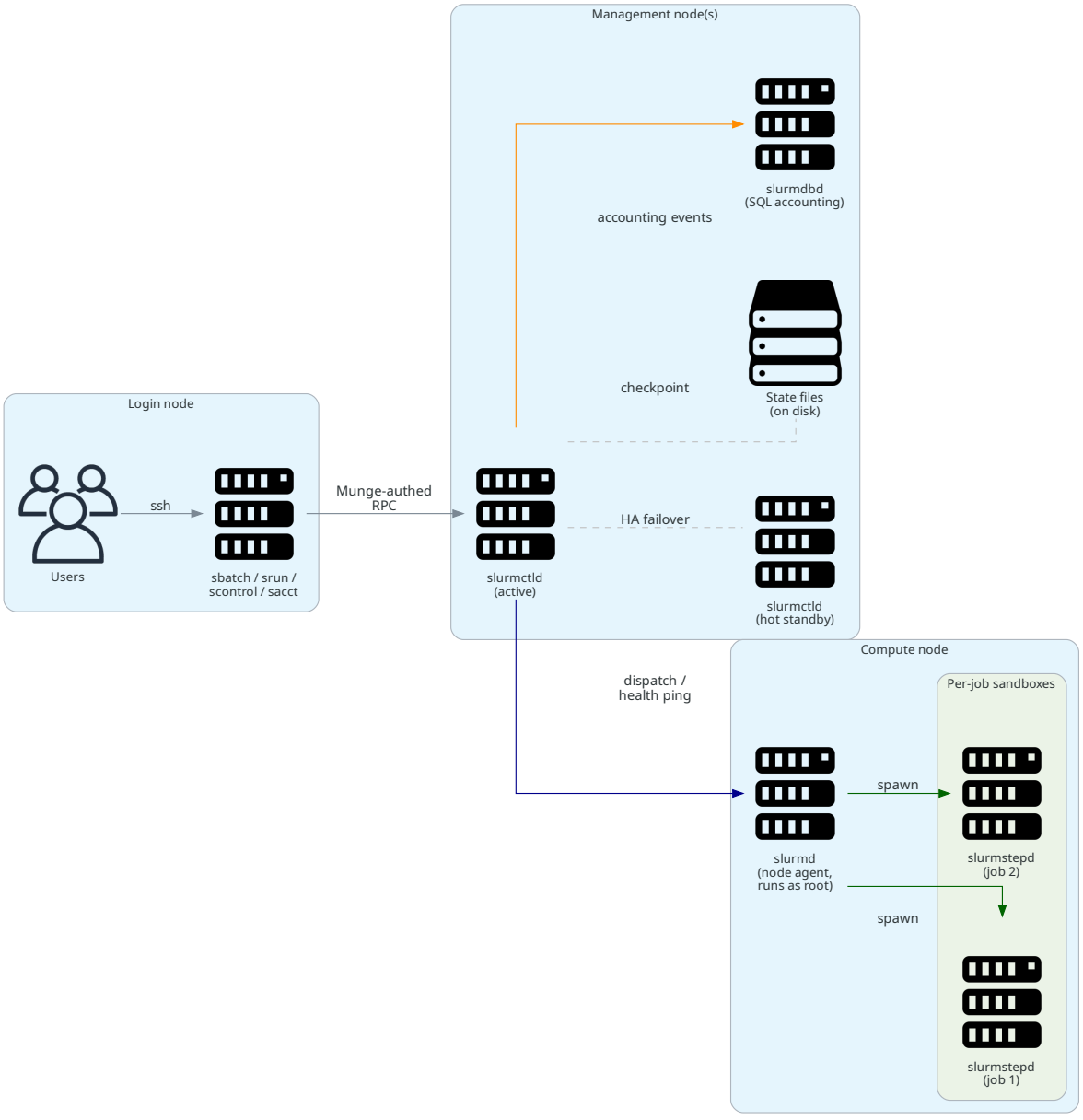


Figure 3: Slurm daemon architecture. User commands (`sbatch`, `srun`, `scontrol`, `sacct`) leave the login node as Munge-authenticated RPCs into `slurmctld`, the only process that performs scheduling; `slurmctld` keeps cluster state in memory, checkpoints it to local state files, and a hot-standby `slurmctld` can take over within seconds if the primary fails. The controller dispatches work to `slurmd` on each compute node (along with regular health pings), and `slurmd` spawns a short-lived `slurmstepd` per job to execute the user's code inside its cgroup sandbox. Accounting events flow from `slurmctld` to `slurmdbd`, which persists them in a SQL database used later by `sacct` and the fair-share framework. `munged` runs on every node with a copy of the shared key and signs every inter-daemon RPC, which is the basis of the trust model. Generated with the `diagrams` Python library; source at `scripts/Slurm_daemons.py`.

directory) and then executes the user's program inside it. Isolating this work in a separate process ensures that a crashing user job terminates only `slurmstepd`; `slurmd` itself survives and continues to serve other jobs.

slurmdbd: The accounting daemon - `slurmdbd` is an optional helper that connects `slurmctld` to a SQL database (usually MariaDB or MySQL). It records which user submitted which job, when it started, how long it lasted, and how many CPU- and GPU-hours it consumed. Operators inspect this history with commands such as `sacct` and `sreport`. The same database is also where Slurm stores fair-share information that affects future scheduling decisions.

Plugins - Most of Slurm's behavior is implemented as *plugins*, small shared libraries that the controller loads at startup. This is how one Slurm binary can operate with different schedulers, different authentication backends, or different GPU support without recompilation. The plugin families that matter for this thesis are the scheduler (FIFO or backfill), the GRES plugins (GPUs, NICs), and the authentication plugin (`auth/munge`). Sites usually customize Slurm by writing small plugins or Lua scripts rather than patching the source.

2.3.2 Configuration and authentication

Slurm is configured through a handful of plain-text files. The main one is `slurm.conf`, a single key-value file that describes the whole cluster: the controller's hostname, the list of compute nodes and their hardware (`NodeName=node01 CPUs=64 RealMemory=256000 ...`), the partitions (queues) and their policies, the choice of scheduler and authentication plugin, timeouts, log locations, and so on. A few smaller companion files cover narrower topics (`gres.conf` lists per-node GPUs and other generic resources, `cgroup.conf` controls how cgroup isolation is applied), but `slurm.conf` is the cluster's source of truth.

Every node in the cluster has to hold an identical copy of this file. When `slurmd` registers, the controller compares a hash of the node's `slurm.conf` with its own and logs an error if they differ, because mismatched files cause failures that are hard to trace. Distributing the configuration therefore matters in practice: An operator cannot simply edit the file in one place. At GWDG, Warewulf delivers the files to the nodes (Section 2.2).

Configless mode - Newer Slurm versions can operate in *configless* mode: A node needs only the controller's hostname, and `slurmd` then fetches `slurm.conf` and its companions directly from the controller at startup. This matters for the thesis, because a re-imaged node needs no site-specific configuration inside its image. The same Warewulf image serves any node, and a later edit to the configuration reaches the node automatically the next time it boots or receives a reconfigure instruction.

Authentication via Munge - Slurm implements no authentication of its own; it delegates that to *Munge* (MUNGE Uid 'N' Gid Emporium), a small daemon that runs alongside `slurmd` on every node. Munge is simple: Every node holds a copy of the same secret file of random bytes (the *Munge key*, usually `/etc/munge/munge.key`). When one Slurm daemon wants to send an RPC to another, it asks the local `munged` to wrap the caller's user and group ID into a short signed token using that key. The receiving daemon hands the token to its own `munged`, which verifies the signature and returns the original UID/GID. Because only nodes that hold the shared key can produce valid tokens, the receiver can trust the identity inside.

In practice the Munge key is the cluster's root credential: Anyone who has it can impersonate any user, including `root`, against any Slurm daemon. The key is therefore generated once at cluster setup and distributed only through trusted channels. In this thesis a Warewulf overlay file delivers it at boot.

Applying changes - Editing `slurm.conf` alone has no effect: The running daemons still hold the old configuration in memory. To adopt the change, the operator invokes `scontrol reconfigure` on any node. The controller then tells every `slurmd` in the cluster to re-read its configuration files and merge the new values into its in-memory state, all without dropping the queue or interrupting running jobs. Most everyday changes, such as a new partition limit or a changed timeout, take effect this way.

For this thesis, the negotiation service never modifies `slurm.conf` at runtime. Every node that can ever take part in the cluster, both the always-Slurm nodes and the ones that move back and forth to Kubernetes, is declared once in `slurm.conf` when the cluster is set up. While a node is on the Kubernetes side its `slurmd` is not running, so the controller regards the node as unavailable and schedules no job onto it; the rest of the cluster continues to operate normally. When the node is re-imaged back to its Slurm role and boots, `slurmd` comes up, registers with the controller, and the node becomes available for jobs again, all without a `reconfigure` or any restart on the other nodes. The slow path of editing `slurm.conf` and reloading the cluster is reserved for real configuration changes (new partitions, changed time limits, added hardware), not for routine node movement.

2.3.3 The resource model

Slurm models the cluster as a set of *nodes* and tracks three categories of resource on each node: CPUs, memory, and everything else (GPUs, NICs, licenses, and so on).

CPUs and memory - Each node reports to the controller how many hardware threads it holds (Slurm calls these *CPUs*) and how much RAM is available to jobs. A job requests a number of CPUs and an amount of memory, and Slurm confines it to that slice of the node with cgroups, so two jobs on the same node cannot encroach on each other's cores or memory.

Generic resources (GRES) - Anything that is not a CPU or memory is a *generic resource*: GPUs, FPGAs, fast NICs, scratch disks, software licenses. Each node declares the GRES it owns, and jobs request them with flags such as `--gres=gpu:a100:2`. Slurm then ensures that the job sees only the devices granted to it (for GPUs, by setting `CUDA_VISIBLE_DEVICES`). Sub-types such as `gpu:a100` against `gpu:h100` allow one partition to hold mixed hardware without splitting it across separate queues.

Features - Nodes can also carry free-form labels called *features* (`avx512`, `infiniband`, `nvlink`, and so on) that describe properties Slurm does not track as a count. Jobs filter on them with `--constraint=avx512`, which is the simplest way to express "schedule me only on nodes with property X".

2.3.4 Job submission and the user interface

A user interacts with Slurm through three main client commands.

sbatch - `sbatch` submits a *batch job*. The user writes a shell script whose first lines are `#SBATCH` directives describing the resource request, and `sbatch` hands that script to `slurmctld` for queuing. When the job starts, the script executes on the first allocated node (the *batch host*) inside a Slurm-managed environment; the user's stdout and stderr are written to files the operator configures. The script body typically launches one or more *job steps* through `srun`, each of which may occupy a subset of the surrounding allocation. The resource request is fixed at submission time: A batch job cannot grow or shrink its allocation once it has started. This is why a node cannot leave Slurm in the middle of a job without preempting that job.

srun - `srun` launches a job step. Inside an existing allocation it uses that allocation's resources; outside one it first requests an allocation. It is also the usual launcher for MPI programs.

salloc - `salloc` requests an allocation and starts a (typically interactive) shell against it. It is the usual route to an interactive session on compute nodes for development, profiling, or debugging.

Job arrays and dependencies - A single `sbatch` can submit a *job array* (`--array=1-1000`), many independent tasks that share one script, and a job can declare that it starts only after another job has finished (`--dependency=afterok:12345`). Both features change only how jobs enter the queue; the scheduler treats each array task as a separate job.

2.3.5 Partitions

A *partition* is Slurm's name for a queue. Each partition covers a subset of the cluster's nodes and applies its own policy: a default and maximum time limit, a list of users or groups allowed to submit, a default and maximum number of nodes per job, a `PriorityTier` that governs preemption, an `OverSubscribe` setting that controls whether multiple jobs can share a node, an `AllocNodes` list that restricts which login nodes can submit to it, and so on. A node can belong to several partitions at once.

In production, partitions are how operators offer different service classes on the same cluster. Typical examples are short-debug, standard, long-running, GPU, big-memory, and reserved-for-research-group partitions.

For this thesis, partitions stay static. All compute nodes, both the always-Slurm ones and the dynamic ones that move back and forth to Kubernetes, are listed in a single compute partition when the cluster is set up, and the node list is never edited at runtime (Section 2.3.2). Users need not change their submission scripts: They always submit to the same partition, and Slurm schedules only onto the nodes that are up at that moment.

2.3.6 Scheduling

Slurm operates two schedulers. The *main scheduler* traverses the pending queue in priority order every few seconds and starts any job that fits immediately; if the top job cannot start, the pass stops there, so a lower-priority job cannot overtake it. The *backfill scheduler* [Lif95] then fills the remaining idle gaps with shorter, lower-priority jobs that can finish before the resources reserved for higher-priority jobs are needed.

Job priority itself comes from the `priority/multifactor` plugin, which combines age, fair-share, job size, partition, Quality of Service (QOS), and a few smaller terms into a

weighted sum. Operators choose the weights to match site policy. Slurm also supports preemption, topology-aware placement, and gang scheduling, but none of those features matter for the negotiation service built in this thesis.

2.3.7 Fair-share and accounting

Slurm applies a fair-share priority scheme that raises the priority of under-using accounts and lowers that of heavy users, with QOS levels layering bundles of limits on top (maximum jobs, time ceilings, priority offsets). For this thesis only one point matters: Slurm bills an account only for the jobs it runs, so a node that serves Kubernetes accrues no GPU-hours against any Slurm account.

2.3.8 Node states

A Slurm node carries a base state (`IDLE`, `ALLOCATED`, `DOWN`, and a few others) plus optional flags such as `DRAIN`. The Slurm manual lists them in full; only a few states matter for this thesis.

When the negotiation service hands a node from Slurm to Kubernetes, it first drains the node, so that running jobs finish while no new ones are placed on it, and then sets it `DOWN` and reboots it. In the reverse direction, the node boots a Slurm image, `slurmd` registers with the controller, and the node returns to `IDLE`. The service drives these state changes through `scontrol update`. Because every node is pre-declared in `slurm.conf` (Section 2.3.2), a move never creates or deletes a Slurm node object.

2.3.9 Power saving and cloud nodes

Slurm has a power-saving framework[5] that suspends idle nodes through an operator-supplied `SuspendProgram` and restores them with a `ResumeProgram` when work arrives; the same hooks also drive Slurm's *cloud* node feature, spawning and tearing down cloud VMs on demand. The negotiation service in this thesis generalizes that pattern: Instead of toggling power or VM lifecycle, it switches the node between a Kubernetes image and a Slurm image through Warewulf.

2.3.10 Failure handling and node health

Nodes fail regularly in a production cluster, and Slurm provides several layers of defense against the resulting disruption.

The first layer is `SlurmdTimeout`. If `slurmctld` receives no message from a `slurmd` within this many seconds, it marks the node `NOT_RESPONDING` and, after further retries, `DOWN`. Jobs running on a `DOWN` node are either killed or, if the user submitted them with `--requeue`, put back into the pending queue.

The second layer is the `HealthCheckProgram`, an operator-supplied script that executes periodically on each compute node. A non-zero exit drains the node, with the script's stderr recorded as the drain reason. Sites typically use it to verify that scratch filesystems are mounted, that GPUs respond, and that critical kernel modules are loaded.

The third layer is the `Prolog`/`Epilog` pair, scripts that execute before and after each job on the compute node. A non-zero exit from a prolog drains the node and requeues the job, so checks such as "is `/scratch` writable?" keep broken nodes out of service

[5]Slurm Power Saving Guide: `https://slurm.schedmd.com/power_save.html`, accessed 12 May 2026.

even when they passed the periodic health check. The variants `PrologSlurmctld` and `EpilogSlurmctld` run on the controller instead of the compute node.

2.3.11 Operator interface

An operator works with Slurm mostly through three CLI tools, each of which is a thin wrapper around an RPC into `slurmctld` or `slurmdbd`.

scontrol - `scontrol` is the general administrative tool. It displays and modifies almost any object the controller knows about (nodes, jobs, partitions, reservations, federations), and operators also reload configuration, schedule reboots, or fail over to the backup controller through it. The negotiation service drives the Slurm side of every node transition only through `scontrol update`.

sinfo and squeue - `sinfo` prints node and partition summaries; `squeue` prints the pending and running job queue. Both accept format strings, so scripts can read their output reliably. The negotiation service reads the queue with `squeue` to assess Slurm-side demand and the node states with `sinfo` before deciding which nodes to drain.

sacct and sacctmgr - `sacct` queries the accounting database for completed and running job records; `sacctmgr` manages the association/account/QOS hierarchy. Neither lies on the hot path of the negotiation service, but `sacct` supplied the six-day job trace the evaluation chapter replays as its Slurm workload.

2.4 Kubernetes

Kubernetes[6] is an open-source orchestrator for containerized workloads. It grew out of Google's experience with Borg [Ver+15] and is now the common platform for containerized services such as web back-ends and machine-learning inference. GWDG currently serves LLMs as Slurm jobs through the SAIA stack [Doo+25]. This thesis examines hosting the same workload on Kubernetes instead, on nodes the negotiation service lends to Slurm when inference demand falls and takes back when it rises.

This subsection is organized the same way as the Slurm one: the daemon architecture (Section 2.4.1), how clusters are configured and how identity is bootstrapped (Section 2.4.2), the workload model that users see (Section 2.4.3), the user interface (Section 2.4.4), the scheduler (Section 2.4.5), the node-lifecycle primitives the negotiation service relies on (Section 2.4.6), and the self-healing model (Section 2.4.7). The deployment this thesis builds on, Ephemeral Kubernetes, appears separately in Section 3.5.

2.4.1 Control plane and node architecture

A Kubernetes cluster has two kinds of nodes, sketched in Figure 4. *Control-plane* nodes hold cluster state and decide where workloads execute; *worker* nodes execute the user workloads themselves. Every node is an ordinary Linux machine, and the roles differ only in which daemons are started.

[6]Kubernetes Documentation: Concepts: `https://kubernetes.io/docs/concepts/`, accessed 14 April 2026.

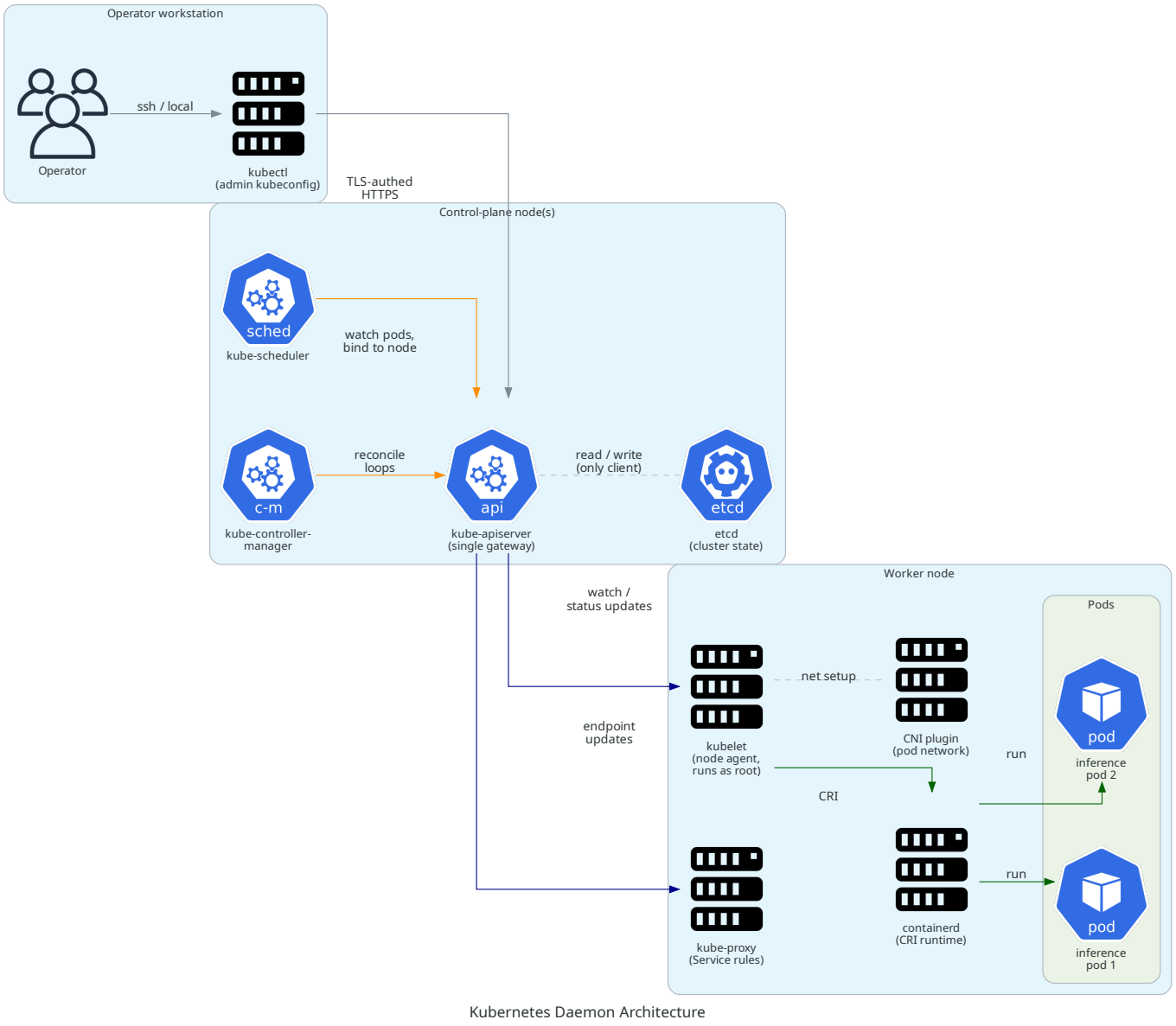


Figure 4: Kubernetes daemon architecture. Operator commands (`kubectl`) leave the workstation as TLS-authenticated HTTPS requests into `kube-apiserver`, the only component allowed to read or write `etcd`, where all cluster state lives. `kube-scheduler` watches the API server for pods whose `spec.nodeName` is empty and binds each one to a node; `kube-controller-manager` hosts the reconciliation loops (Deployment, ReplicaSet, Node, and others) that close the gap between desired and actual state. On every worker, `kubelet` watches the API server for pods assigned to its node, drives the local CRI runtime (`containerd`) to start the containers, and reports node conditions and a periodic `NodeLease` heartbeat back through the API server. `kube-proxy` keeps the kernel rules that route Service virtual IPs to live pods, and the CNI plugin attaches each pod to the cluster network at startup. Unlike Slurm's shared Munge secret, Kubernetes hands out per-component TLS certificates signed by the cluster CA, so a single component can be rotated or revoked on its own. Generated with the `diagrams` Python library; source at `scripts/Kubernetes_daemons.py`.

Control-plane nodes - A control-plane node hosts the API server, etcd, the scheduler, and the controller-manager. A test cluster hosts all four on one machine; a production cluster spreads them across at least three, so that the loss of any one does not disable the cluster. The components execute as *static pods* (manifests the local kubelet reads from disk without consulting the API server), which allows the control plane to recover after a reboot before the API server itself is available. In this thesis the control-plane nodes are fixed (role `k8s-control` in `node-config.conf`, Section 4.2.2), and the negotiation service never moves them.

Worker nodes - A worker hosts the kubelet (the local agent that communicates with the API server), a container runtime (which launches and supervises containers), and `kube-proxy` (which maintains the rules that route Service traffic to the right pod). Workers hold GPUs and host LLM inference replicas. The dynamic workers are the nodes the negotiation service moves between Kubernetes and Slurm: In their Kubernetes role the kubelet runs and the node appears in `kubectl get nodes`; once they are re-imaged for Slurm, the kubelet is gone and the Node object has been deleted.

kube-apiserver - The API server is the only component allowed to read or write etcd. Everything else (controllers, schedulers, kubelets, `kubectl`, the negotiation service) reaches cluster state by sending HTTP requests through it. Because every component shares this gateway, authentication, authorization, audit logging, and rate-limiting are all enforced in one place. The API server is itself stateless and can be scaled horizontally against the same etcd.

etcd - etcd is a small distributed key-value store that holds every piece of cluster state (Pods, Nodes, Services, Secrets, ConfigMaps). It relies on Raft, so fault tolerance requires an odd replica count (three is the usual minimum), and the loss of quorum is the loss of the cluster. For the negotiation service etcd matters indirectly: Every node move (cordon, drain, delete) is ultimately a write to etcd performed by the API server.

kube-scheduler - The scheduler assigns each new pod to a node (Section 2.4.5). For inference pods, the `nvidia.com/gpu` request keeps them on GPU nodes.

kube-controller-manager - The controller-manager is a single process that hosts most built-in reconciliation loops: Deployment, ReplicaSet, Endpoints, Node (which marks nodes `NotReady` when kubelet heartbeats stop), garbage collection, and others. Each loop reads the desired state, reads the actual state, and issues the API calls needed to close the gap. This design lets Kubernetes recover from outages: A restarted controller re-reads the state and continues.

kubelet - The kubelet is the agent on every node, control-plane and worker. It watches the API server for pods assigned to its node, drives the container runtime accordingly, and reports back resource counts (CPU, memory, GPU) and conditions (`Ready`, `MemoryPressure`, `DiskPressure`). It plays the same role on the Kubernetes side that `slurmd` plays on the Slurm side: Its presence defines the node's role.

Container runtime - The kubelet does not start containers itself; it delegates that work to a *container runtime*, usually `containerd` or `CRI-O`, through the CRI interface. The runtime pulls images, prepares cgroups and namespaces, mounts volumes, and supervises lifecycles. Separating kubelet and runtime allows one Kubernetes binary to drive many runtimes, much as Slurm's plugin architecture allows one `slurmctld` to drive many schedulers.

kube-proxy - `kube-proxy` runs on every worker and maintains the kernel rules (iptables, IP Virtual Server (IPVS), or eBPF) that translate Service virtual IPs to one of the backing pods. The Service abstraction depends on it: When a pod is added or removed, the local `kube-proxy` rewrites the rules so later connections find the new backend set.

The CNI plugin - The kubelet does not configure pod networking either. When a pod starts, the kubelet calls a CNI plugin (Calico, Cilium, Flannel, and so on) that allocates an IP, prepares the pod's network namespace, and configures routing so that any pod can reach any other. In Decker and Kunkel's Ephemeral Kubernetes setup [DK25] the CNI is bundled into the worker image, so a freshly booted node has full pod connectivity as soon as `kubeadm join` returns.

2.4.2 Configuration and authentication

Kubernetes is configured through the API server. No global file comparable to Slurm's `slurm.conf` has to remain identical on every node. Cluster state lives in etcd; a node needs only enough local configuration to locate the API server, prove its identity, and start its kubelet. This makes Kubernetes a good fit for Warewulf overlays: A small per-boot bundle suffices for a node to join.

kubeconfig - A *kubeconfig* is a small YAML file that tells a Kubernetes client how to reach the cluster's API server. It bundles three elements: the API server's network address, an identity to authenticate as (a human user, an in-cluster service account, or a node's kubelet), and the credentials that prove that identity (a client certificate or a bearer token). Two kinds of kubeconfig matter for this thesis. The first is the *admin kubeconfig*, used by the human cluster administrator (the *operator*): `kubeadm init` generates it on the leader control-plane node and writes it to `/etc/kubernetes/admin.conf`, and the operator typically copies it onto a workstation at `~/.kube/config` so that `kubectl` finds it by default. The second is the *kubelet kubeconfig*, used by the per-node Kubernetes agent (the kubelet, introduced in Section 2.4.1): `kubeadm join` writes it to `/etc/kubernetes/kubelet.conf` on each worker at boot, after the worker swaps a short-lived bootstrap token with the API server for a long-lived client certificate signed by the cluster CA. In Ephemeral Kubernetes [DK25] the admin kubeconfig has to be reachable from places other than the leader node. The leader's install script copies it from `/etc/kubernetes/admin.conf` into the shared NFS mount at `/share/Kube.config` through a Warewulf overlay (the *admin overlay*), so that worker nodes can present it as a join discovery file and the negotiation service on the Warewulf host can read it to drain nodes, delete stale Node objects, and check cluster health. Figure 5 shows the resulting distribution.

Cluster PKI - Component authentication is based on TLS. `kubeadm init` generates a self-signed root CA (the *cluster CA*), which signs a client certificate for every component (kubelets, controller-manager, scheduler, kube-proxy, operator clients) and the API server's own serving certificate. The CA plays the role Slurm's Munge key plays as cluster root of trust, but Kubernetes issues per-component certificates instead of a single shared secret, so a single component can be rotated or revoked on its own. Certificates issued by `kubeadm` last one year; the kubelet renews its own client certificate automatically.

Bootstrap tokens - A fresh worker has no certificate yet, so it cannot authenticate with TLS on first contact. Kubernetes solves this with a *bootstrap token*: a short-lived shared secret that the worker presents during `kubeadm join`. The API server accepts the

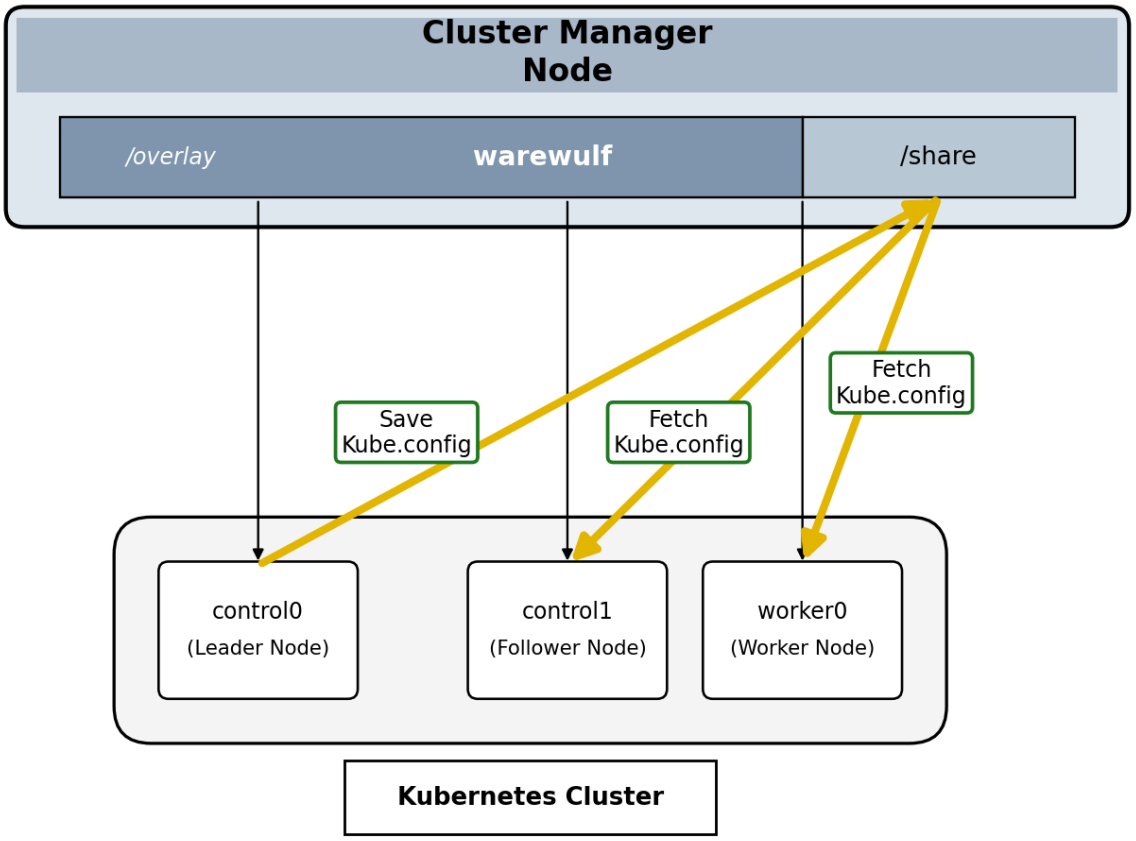


Figure 5: Admin-kubeconfig distribution in Ephemeral Kubernetes. The cluster manager (the Warewulf host) holds two filesystem locations relevant to this flow: the `/overlay` machinery, through which Warewulf delivers per-node configuration, and the `/share` NFS mount, attached to it. The leader control-plane node (`control0`) executes `kubeadm init` and writes the admin kubeconfig onto `/share` as `Kube.config`. From there, the follower control-plane node (`control1`) fetches it to bootstrap itself into the existing cluster, and each worker (`worker0`) fetches it as the `--discovery-file` argument to `kubeadm join` so the kubelet can register a Node object. The thin black arrows show the Warewulf overlay reaching every node at boot; the thick gold arrows show the `Kube.config` Save (leader to `/share`) and Fetch (`/share` to every other node) operations. Generated with `matplotlib`; source at `scripts/Kubernetes_authentication.py`.

token, the cluster signs a client certificate for the worker's kubelet, and `kubeadm` writes it into the node's kubeconfig. The token expires after a set time, 24 hours by default.

Service accounts and RBAC - Workloads inside the cluster authenticate through *service accounts*, whose tokens the API server mounts into pods (a projected, auto-rotating volume in newer releases). Permissions are granted through Role-Based Access Control: A `Role`/`ClusterRole` lists verbs (`get`, `list`, `watch`, `patch`, `delete`) per resource kind, and a `RoleBinding`/`ClusterRoleBinding` grants that role to a user or service account. The negotiation service runs on the Warewulf host with an operator kubeconfig and cluster-admin privileges. If it were ever moved into the cluster as a pod, its service account would be narrowed to cordon/drain/delete on `nodes` and list on `pods`.

Joining and leaving a cluster - A Kubernetes worker's lifecycle is two commands. `kubeadm init` bootstraps the control plane and prints a join command; `kubeadm join` connects a worker through a bootstrap token and CA hash, after which the kubelet registers a Node object and starts receiving pods. Leaving is the reverse: `kubectl drain`, `kubectl delete node`, then stop the kubelet. Every Kubernetes-bound transition in this thesis runs `kubeadm join` from a freshly booted Warewulf image. The node obtains its join credentials with a per-node key that is regenerated for each transition and delivered through the node's overlay instead of being built into the image (Section 4.2.4). The trust model is analyzed in the companion report [Mad26].

2.4.3 The workload model

Where Slurm operates on jobs that run to completion, Kubernetes operates on long-lived declarative *objects* that describe a desired state. Built-in controllers continuously reconcile the actual state with that description. Kubernetes holds no notion of finishing a job, only of declaring, or withdrawing, workloads that should remain alive.

Pods - A pod is the smallest schedulable unit. It groups one or more containers that share a network namespace, volumes, and lifecycle. Most pods hold one application container plus zero or more sidecars (logging, proxy, credential helper). Pods are designed to be disposable: When something goes wrong, the correct response is to delete the pod and let a controller create a new one. The LLM-serving workload places one inference server per pod, bound to one or more GPUs on its host node.

Deployments and ReplicaSets - Users rarely create pods directly. Instead, they create a `Deployment` that declares "N replicas of this image, configured this way". The Deployment controller creates a `ReplicaSet`, which creates the pods and recreates any that crash. Updating the image triggers a controlled rollout: New pods start, become ready, and old ones terminate while a configurable minimum remains alive. If readiness fails, the rollout halts and can be rolled back. The LLM-serving Deployment relies on this mechanism for replica management and upgrades.

Services and Ingress - Pod IPs change on every restart, so clients address `Service` virtual IPs (ClusterIPs) and DNS names instead. `kube-proxy` steers each connection to a healthy backend. Service types are `ClusterIP` (in-cluster only), `NodePort` (also on a fixed port per node), and `LoadBalancer` (external, usually through a cloud or MetalLB). For HTTP, an `Ingress` configures a reverse proxy (NGINX, HAProxy, Traefik) so the cluster is reachable by host name.

ConfigMaps and Secrets - Configuration and credentials live in `ConfigMap` and `Secret` objects rather than baked into images, and they are mounted into pods as files or environment variables. Secrets are stored in etcd, optionally encrypted at rest, and gated by RBAC. An inference Deployment, for example, can keep its model name and access tokens here, so the same image can serve different models.

Namespaces - A namespace is a name-scope inside a cluster. Most object kinds live in one, and RBAC rules are usually namespace-scoped, which makes a namespace a soft tenant boundary (for example, full control of `llm-serving` without touching `kube-system`). Cluster-wide objects (Nodes, PersistentVolumes, ClusterRoles) live outside any namespace.

Other workload kinds - Other abstractions exist but are not on the negotiation-service path: `StatefulSet` (stable identities and per-replica storage, for databases and queues), `DaemonSet` (one pod per node, for log shippers and CNI), `Job` (run-to-completion, the closest analogue to a Slurm batch job), and `CronJob` (Jobs on a schedule). The LLM-serving workload is a Deployment.

Resource requests and limits - Every container declares CPU and memory `requests`, on which the scheduler checks fit, and `limits`, which the kubelet enforces through cgroups. Their combination sets the pod's QoS class: `Guaranteed` (requests=limits, evicted last), `Burstable` (requests below limits), and `BestEffort` (no requests, evicted first). Inference pods would typically be declared Guaranteed, so that the kubelet evicts them last under memory pressure.

GPUs as resources - GPUs are not first-class resources like CPU and memory; the device-plugin interface exposes them. The NVIDIA device plugin (a DaemonSet) reports to the kubelet how many GPUs each node holds, and the kubelet publishes this on the API server as `nvidia.com/gpu`. A pod requests GPUs through `nvidia.com/gpu: N`, and the scheduler treats the request as another filter-pass constraint. This is the Kubernetes counterpart of Slurm's `--gres=gpu:...`, and it keeps inference pods on GPU nodes.

2.4.4 Submission and the user interface

Operators work with Kubernetes through one CLI and one file format.

kubectl - `kubectl` is the standard CLI. Every subcommand (`get`, `apply`, `delete`, `describe`, `exec`, `logs`, `port-forward`, `cordon`, `drain`, and so on) is translated into HTTP requests against the API server using the active kubeconfig. `kubectl logs`, `exec`, and `port-forward` are convenient debugging tools. The negotiation service addresses the same API (cordon, drain, and delete on Node objects) by invoking these subcommands from the Warewulf host (Section 4.2.7).

Manifests - A manifest is a YAML or JSON file that describes one or more objects through `apiVersion`, `kind`, `metadata`, and `spec`. It plays the role of a Slurm batch script: The user writes it, invokes `kubectl apply -f manifest.yaml`, and the cluster keeps the objects alive. `apply` is idempotent and merge-aware: Re-applying the same file is a no-op, and an edited file produces only the diff.

Declarative versus imperative - Imperative commands (`kubectl create deployment llm --image=registry/llm-server:1.0`) are convenient for one-off work. Declarative `apply -f` keeps cluster state in version control and makes upgrades reviewable, and tools such as Helm or Argo CD build on it.

Higher-level packaging - Real applications span many objects (Deployment, Service, ConfigMaps, Secrets, HPA, Ingress). *Helm* packages templated manifests as a *chart* with per-environment values; *Kustomize* (built into `kubectl` through `apply -k`) layers overlays on a base set of manifests. Neither lies on the negotiation-service path, which touches only Node objects.

2.4.5 Scheduling

The scheduler watches the API server for pods with empty `spec.nodeName` and assigns each one in two passes.

Filter pass - The pass discards a node if it cannot host the pod at all: not enough CPU, memory, or GPU for the `requests`, a mismatched `nodeSelector`, an unmatched taint, forbidden affinity, or unavailable host ports. Only feasible nodes survive.

Score pass - The survivors are ranked by priority functions: image already cached, least-loaded, soft-affinity match, replica spread. The highest-scoring node is written into the pod's spec, and the local kubelet then starts the pod.

Taints and tolerations - Taints (`key=value:effect`) keep workloads off particular nodes. The effects are `NoSchedule`, `PreferNoSchedule`, and `NoExecute` (which also evicts existing pods that do not have a matching toleration). Cordoning a node (Section 2.4.6) adds such a `NoSchedule` taint, which is how the negotiation service keeps new pods off a node it is about to move.

Node affinity - Where taints push pods away, node affinity pulls them in. A pod declares required or preferred node labels (for example `accelerator=nvidia-a100`), so that inference pods reach only A100 nodes.

Pod affinity and anti-affinity - Pods can also declare preferences about other pods. Affinity packs related pods together; anti-affinity spreads them out. An inference Deployment can declare anti-affinity so that two replicas of the same model never share a node; reclaiming one node then removes at most one replica of each model.

Priority and preemption - A pod's `priorityClassName` allows the scheduler to evict (*preempt*) lower-priority pods to free room for higher-priority ones. The negotiation service relies on no preemption; its guarantees rest on accurate resource requests and on explicit cordon and drain operations.

2.4.6 Node states and draining

The node lifecycle is the Kubernetes layer most directly relevant to the negotiation service: Every role transition goes through the Node object's state machine, just as every Slurm-side transition goes through the `slurmd` state machine in Section 2.3.8.

The Node object - When a worker joins, the kubelet creates a Node object that carries the node's name, capacity (`cpu`, `memory`, `nvidia.com/gpu`), labels, taints, and addresses. The object also carries *conditions* (`Ready`, `MemoryPressure`, `DiskPressure`, `PIDPressure`, `NetworkUnavailable`) that the kubelet sets, plus a `NodeLease` that is refreshed every few seconds as a lightweight heartbeat.

Heartbeats and the unhealthy timeout - If heartbeats stop, the controller-manager waits `node-monitor-grace-period` (default 40 s) before marking the node `NotReady`, and after a further five minutes by default it evicts the node's pods so controllers can recreate them. This is roughly the Kubernetes counterpart of Slurm's `SlurmdTimeout`.

cordon - `kubectl cordon` *`node`* sets `Unschedulable`, which the scheduler respects in its filter pass. Existing pods keep running; cordon evicts nothing on its own. It is non-destructive and reversible through `kubectl uncordon`, and it is the natural first step when reclaiming a node.

drain - `kubectl drain` *`node`* cordons the node and evicts its pods through the eviction endpoint (`/pods/<name>/eviction`), which checks each eviction against any `PodDisruptionBudget` the workload owner has declared. If a budget would be violated,

the eviction is denied and `drain` retries, so the owner controls how aggressively the cluster can evacuate their pods. Once the drain completes, the node holds no application workload and can be powered down or re-imaged.

Deleting a Node - Cordon and drain leave the Node object in place. `kubectl delete node` *`name`* removes it from etcd. A node returning from a Slurm role has booted a fresh image without any kubelet credentials, so it runs `kubeadm join` again and registers a new Node object.

The transition path used by the negotiation service - A node leaving Kubernetes is cordoned, drained, and deleted as a Node object before Warewulf re-images it; a node entering Kubernetes runs `kubeadm join` and registers a new Node object. Section 4.2.4 gives the full sequence in both directions.

2.4.7 Failure handling and self-healing

Kubernetes treats failure as routine and recovers through layered controllers. Inside a container, the kubelet applies liveness, readiness, and startup probes: Liveness restarts a stuck container, readiness withdraws a pod from the Service endpoints until it can serve (the important one for LLM pods, whose model weights can take minutes to load), and startup suppresses liveness during slow first boots. Restart policies govern in-place recovery, and repeated crashes produce CrashLoopBackOff back-off.

Above the container level, ReplicaSets recreate pods that disappear with their node, so the negotiation service can drain and delete a node without losing replicas for good. PodDisruptionBudgets cap voluntary evictions through `minAvailable` or `maxUnavailable`, so a drain stalls before it removes more replicas than the owner allows. Node failure (kernel panic, network partition, or an unplanned reboot) flows through the unhealthy timeout, controller-manager eviction, and scheduler replacement. The HorizontalPodAutoscaler, which scales a Deployment's replica count with load, is off the negotiation path; the service reads the demand directly (Section 4.2.3).

3 Related Work

The problem of hosting cloud-style and batch-style workloads on the same hardware has been studied for over ten years, and the systems that address it occupy different points on a spectrum. At one end are projects that keep the two resource managers separate and move nodes between them. At the other end are projects that place Kubernetes inside a Slurm job, or Slurm inside a Kubernetes pod, so that only one scheduler ever owns the hardware. This chapter presents the seven earlier systems closest to this thesis: Liu et al.'s Balancer for capacity negotiation (Section 3.1), the Torque-Operator for cross-scheduler container submission (Section 3.2), High-Performance Kubernetes (HPK) for embedding Kubernetes in HPC jobs (Section 3.3), Kind Slurm Integration (KSI) for building Kubernetes clusters inside Slurm jobs (Section 3.4), Ephemeral Kubernetes for image-based Kubernetes provisioning on HPC hardware (Section 3.5), SAIA for hosting real-time AI services through Slurm (Section 3.6), and the inference-scaling system of Trappen et al. that fronts Slurm-hosted vLLM with Kubernetes (Section 3.7). For each system, the section states what it does and then identifies the gap that motivates this thesis. Ephemeral Kubernetes receives more space than the others, because the negotiation service builds on it and later chapters name its components. Section 3.8 then turns to a different strand of earlier work, forecasting a workload from its own history, and to the model the predictive policy relies on. Section 3.9 closes with a summary table that assesses each system against the four requirements the negotiation service has to meet: dynamic node reassignment, bare-metal performance, Kubernetes-native service abstractions, and Slurm compatibility.

3.1 The Balancer

Liu et al. [Liu+18] address the same problem as this thesis: One institution operates two clusters with different schedulers, and both remain under-used, because nodes sit idle in one while the other has work waiting. Their setup is an experimental-science cluster at Argonne National Laboratory that operates an OpenStack on-demand pool alongside a Torque batch pool. The on-demand side serves time-critical analysis of detector output; the batch side executes longer computations that can wait. With no coordination, each pool has to be sized for its own peak, so the institution pays for hardware that is rarely fully used.

Their solution, the Balancer, is a service that operates alongside the two resource managers and continuously re-examines the boundary between the pools. The Balancer never schedules jobs itself; it only moves nodes. Each pool has a resource manager (an on-demand manager backed by OpenStack and an on-availability manager backed by Torque), and the Balancer communicates with each through a small interface for requesting, releasing, reclaiming, and querying nodes. Reassignment is non-disruptive: A node only joins the on-demand pool after its current batch job finishes, so no running work is killed or checkpointed. Liu et al. propose three policies. A *basic* algorithm tops up on-demand availability with a small reserve; a *hint* algorithm draws on an advance-notice signal from users who can predict their need 15–30 minutes ahead; and a *predictive* algorithm forecasts demand from past usage when no user hints are available. Replaying two years of Argonne traces shows that the system can cut the fixed on-demand allocation by 82% while serving the same set of requests, and can handle bursty on-demand workloads up to 10% of total cluster capacity without disturbing batch throughput [Liu+18].

Gap - The Balancer is the closest earlier system to the negotiation service in this thesis, and several of its design choices carry over directly: a separate centralized service, asymmetric reservation and release, and parameter-driven policy. The differences lie in the layers below. Liu et al. move nodes between OpenStack, which hands them to on-demand users as virtual machines, and Torque, and they move a node by enabling or disabling it in each resource manager, without reprovisioning it. This thesis moves nodes between Kubernetes and Slurm on bare metal, where every move re-images the physical machine. The inference users at GWDG also send no advance hints like those of the on-demand users at Argonne, so the policies in this thesis work only from observed load and, in the predictive case, a forecast of it. Finally, neither cluster in their setup hosts LLM inference, so latency-sensitive service traffic and GPU draining are not part of their evaluation.

3.2 The Torque-Operator

Zhou et al. [Zho+21] take a different approach to the same problem: Rather than moving nodes between schedulers, they allow a user to submit work to one scheduler and let the other execute it. Their setup has two long-lived clusters, a Kubernetes cluster for cloud workloads and a Torque cluster for HPC jobs, with a shared login node that belongs to both. The Torque-Operator is a Kubernetes operator that adds a `Torquejob` custom resource. A user describes a containerized HPC job in a Kubernetes YAML file, the operator creates a dummy pod that holds the description, and a small gRPC proxy called *red-box* forwards the job to Torque through `qsub`. Torque then executes the job on its compute nodes through Singularity, and a second pod in Kubernetes collects the results when the job finishes. The architecture builds on earlier work on the WLM-Operator, which provided the same capability for Slurm, by replacing Slurm interactions with Torque interactions and by adding the gRPC proxy as a uniform translation layer.

Gap - The Torque-Operator solves a real need (a Kubernetes user who wants to launch an HPC job without leaving their cloud workflow) but it does not solve the problem this thesis targets. Both clusters stay fully provisioned at all times, so an idle Kubernetes worker remains a Kubernetes worker even when Torque has a long queue; the hardware split is fixed. The operator is also a bridge, not a co-location mechanism: The real HPC job still executes as a Torque job under Singularity, with all the data-staging overhead of moving inputs and outputs across the boundary. Finally, the system is built around Torque, which Slurm has largely replaced in production HPC deployments; the GWDG cluster operates Slurm, so the operator's machinery would require a rewrite before deployment. The negotiation service in this thesis instead leaves Kubernetes-native applications on Kubernetes, without translation, and Slurm jobs under Slurm, without containers in the request path, while moving the underlying hardware between them.

3.3 High-Performance Kubernetes

Chazapis et al. [Cha+23] tackle a closely related problem from the opposite direction: Rather than bridging two long-lived clusters, they put a Kubernetes runtime inside an HPC cluster so that only one scheduler is in charge of the hardware. Their system, High-Performance Kubernetes (HPK), hosts per-user Kubernetes "mini Clouds" as user-level services on an ordinary Slurm cluster. Each user deploys a private HPK instance that exposes the Kubernetes API; behind that API, pod lifecycle commands are translated into Slurm `sbatch` submissions and execute under Singularity/Apptainer as ordinary user

processes. HPK supports almost the entire Kubernetes object model (pods, deployments, services, jobs, volumes), and the authors show Argo Workflows, the Spark operator, and TensorFlow Serving running unchanged on top of it. In Wickberg's taxonomy of Slurm–Kubernetes integration [DMK25], HPK is an *adjacent* solution: Slurm and Kubernetes coexist on the same machines and cooperate through a bridging tool, but their responsibilities remain separate.

Gap - HPK is a natural fit for a multi-tenant HPC site because each user can raise their own Kubernetes view of the cluster without root and without administrator intervention. That is also its limit for the use case in this thesis. HPK never owns a node: Every pod is a Slurm job, every service is a Slurm allocation, and Kubernetes networking primitives such as `NodePort` are unavailable, because they require control of physical interfaces. An LLM inference service at GWDG that needs low latency would have to place its pods in Slurm allocations, which carry the batch-style wait-time variance that motivates SAIA (Section 3.6) and the thesis itself. HPK also does not move hardware between paradigms; it shares hardware by virtualizing one paradigm on top of the other. The negotiation service in this thesis treats Kubernetes as a first-class scheduler on real nodes, accepts the operational cost of doing so, and moves the node-paradigm boundary in response to demand.

3.4 Kind Slurm Integration

Decker et al. [DMK25] extend the survey of Slurm–Kubernetes integration with a system that sits at the opposite end from HPK. Kind Slurm Integration (KSI) follows what Wickberg calls the *over* model: A whole Kubernetes cluster is built inside a single Slurm job and torn down when the job ends. KSI relies on rootless Kind (Kubernetes-in-Docker) on top of rootless Podman, so the Kubernetes control plane executes as ordinary container processes inside a user's Slurm allocation, with no kernel privileges and no help from HPC administrators. The paper compares KSI against Bridge Operator, HPK, and kube-slurm on six axes (functional support, ease of use, maintenance, startup time, CPU and memory performance, and network performance) and finds that KSI is the only system among them that offers the full Kubernetes feature set inside a Slurm job, at the cost of slower startup and lower network throughput than bare-metal alternatives [DMK25].

Gap - Between them, KSI and HPK cover the case where a single Slurm user wants Kubernetes semantics for the length of a batch job. They do not cover the case this thesis addresses: a long-running, multi-tenant LLM inference service that has to share a physical cluster with batch workloads while keeping bare-metal performance. KSI's Kubernetes cluster lives only inside one Slurm job, so every service deployment is capped by the job's wall-clock limit and runs inside nested rootless containers. The network penalty the KSI paper reports is acceptable for short experiments but a poor fit for a service that has to answer requests quickly for months. The negotiation service in this thesis again takes a different position: Kubernetes operates on bare metal across many nodes for as long as demand requires, and the boundary against Slurm moves with that demand rather than disappearing when a Slurm job ends.

3.5 Ephemeral Kubernetes

Decker and Kunkel [DK25] take yet another approach: Rather than placing Kubernetes inside Slurm or bridging between them, they operate a full Kubernetes cluster directly on the HPC site's hardware through the same stateless provisioning system (Warewulf) that already provisions Slurm nodes. The control plane (`kubeadm`, etcd, scheduler, API server) and the worker plane (`kubelet`, container runtime, CNI plugin) live inside Warewulf images. Booting a node into Kubernetes is a PXE boot with the `k8s-control` or `k8s-base` image; tearing down the cluster is a power cycle that returns the node in another role.

The subsections below present its components, the cluster-formation flow, and the credentials that hold the cluster together, because later chapters refer to them by name. A companion security analysis [Mad26] audits that credential model and proposes two architectural enhancements; the negotiation service takes the secured variant as its deployment target.

3.5.1 Motivation and approach

Industry typically deploys Kubernetes on long-lived virtual machines: The cluster is configured once, workers are added over time with `kubeadm join`, and the control plane and etcd state persist for months or years. HPC sites cannot adopt that model directly. Hardware is shared between batch and service roles, leftover state from a previous role (iptables rules, mounted filesystems, container runtime state) is a security liability, and the cluster has to appear and disappear on demand instead of persisting indefinitely.

Decker and Kunkel address this by treating Kubernetes itself as a stateless workload delivered through Warewulf [DK25]. Nodes boot diskless from network-served images, hold only the software their current role requires, and are wiped clean on reboot, so a node can leave Kubernetes and boot into another role with no leftover state. They report a deployment time of under 90 seconds for a five-node Kubernetes cluster starting from a cold boot.

3.5.2 Architecture and components

Ephemeral Kubernetes ships as two Warewulf images plus one shared system overlay, backed by an NFS share that the Warewulf host serves to all nodes.

Two images - The *control image* holds the Kubernetes control-plane binaries (`kube-apiserver`, `kube-scheduler`, `kube-controller-manager`, `etcd`) together with HAProxy and Keepalived for high availability. The *base image* holds only the kubelet and container runtime and serves worker nodes. Both images bundle pre-pulled container images, so that the cluster can form without contacting any registry on the public internet, which matters for air-gapped or otherwise restricted HPC environments. Figure 6 shows how the pieces of a control (leader) node fit together and which of them touch the NFS share.

Shared system overlay - A single Warewulf system overlay (`k8s-overlay`) is layered on top of either image at boot. It carries the installer script, the Phylactery service files, and the kubeconfig and certificate paths that the install script expects. The same overlay is used for control and worker roles; the install script branches on the node's hostname (matching `control`N or `worker`N) to decide whether the local node follows the control-plane initialization path or the worker join path.

NFS share - A shared NFS mount at `/share` on every node serves four purposes in the original architecture. It keeps configuration synchronized between nodes, holds the pre-loaded container image archives the install script imports, coordinates leader election during bootstrap, and distributes the cryptographic credentials the cluster requires to form. The same Warewulf cluster manager exports this share. Section 3.5.6 explains why the last role is the architecture's main attack surface.

Virtual IP and Keepalived - The control plane is reachable through a single *virtual IP* (VIP), a floating address that always points at one of the live control-plane nodes. Keepalived, a small daemon on every control-plane node, owns the VIP: It elects one node as the holder of the VIP through VRRP and moves the VIP to a standby node within seconds if that node fails. Clients (kubectl users, joining workers, the negotiation service) address only the VIP; they never need to know which physical node currently answers it.

HAProxy - On every control-plane node, HAProxy listens on the VIP-bound port and load-balances incoming requests to the live `kube-apiserver` instances on all control-plane nodes. Together, Keepalived (which decides which node owns the VIP) and HAProxy (which decides which API server answers a request once it arrives on a node) provide the cluster with a highly available control-plane endpoint without an external load balancer. This is convenient for the negotiation service: The kubeconfig it holds on the Warewulf host points at the VIP, so control-plane membership changes require no reconfiguration on the service's side.

Phylactery - Phylactery is a Python daemon (`phylactery.py`) delivered in the system overlay and started as a systemd service on every control-plane node. Worker nodes carry no Phylactery. It exists because the static HAProxy configuration alone cannot follow control-plane membership changes, and because Warewulf's stateless reboots break the standard rejoin flow. Section 3.5.4 presents its two purposes.

Install script - The install script `k8s-install.sh` is invoked once per boot from a oneshot systemd unit. It performs the network and kernel configuration that cannot be baked into the image (because the Docker build behind Warewulf boots no kernel), imports the pre-loaded container archives from `/share`, races for leadership, and then branches into either the leader path (`kubeadm init`, deploy the network plugin, upload credentials) or the follower or worker path (wait for the leader, fetch credentials, `kubeadm join`).

Other components - Kubeadm, kubelet, containerd, and the standard kubectl/etcdctl/helm CLIs come pre-installed in the base image. Flannel is the CNI plugin in the reference setup, but any plugin available from pre-loaded container images or from the internet is equally suitable.

3.5.3 Cluster formation and node join

Decker and Kunkel describe the deployment as a single parallel process across all nodes. The steps below are those the install script performs for each role; Figure 7 shows the same flow as a sequence diagram, with the leader, follower, and worker swimlanes called out separately.

Boot and image fetch - Control nodes request the control image from Warewulf, worker nodes request the base image. Both apply the shared `k8s-overlay` on top. Once systemd is up, the install script executes as a oneshot service.

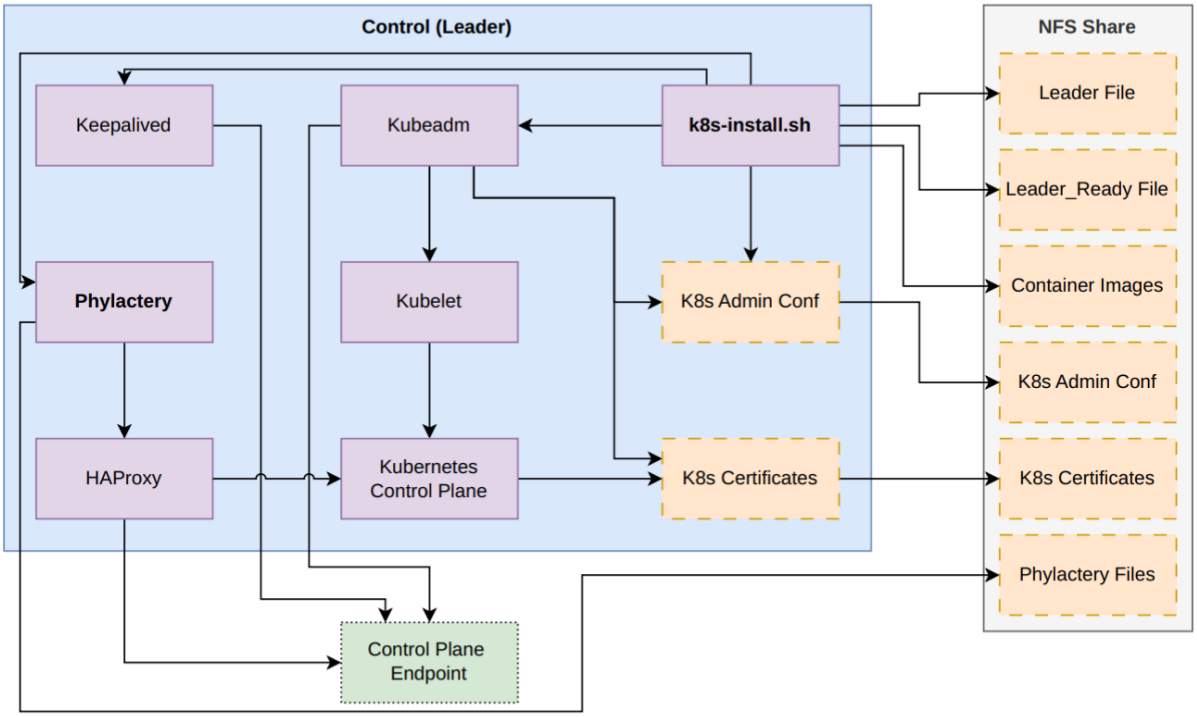


Figure 6: Component diagram for Ephemeral Kubernetes control leader node and NFS share. Components with purple background and solid lines are executable programs, light orange backgrounds and dashed lines indicate files and green background and dotted lines represent network abstractions. Components with names in bold have been developed as part of this research effort. Taken from the original Ephemeral Kubernetes paper [DK25].

Container-image import - Every node imports the pre-loaded container archives from `/share` into its local container store. This step takes several seconds, and small differences in NFS read speed spread the nodes' timing apart before the next step.

Leader election - All control-plane nodes attempt to create the same file, `/share/leader`, containing their own hostname. The first node to succeed becomes the *leader*; the rest become *followers*. The two roles are those of Figure 5: The leader is the node that later executes `kubeadm init` and saves the admin kubeconfig to `/share`, and the followers are the control-plane nodes that fetch it from there. Because the import step spreads the nodes' timing, the race produced a single leader in every experiment the paper reports. The authors note that two nodes could in principle both succeed before the NFS synchronizes, and they suggest a timeout-based variant or Raft-based consensus as more robust alternatives [DK25].

Leader path - The leader starts the Phylactery service and Keepalived, then executes `kubeadm init` to create the cluster. This generates the Public Key Infrastructure (PKI) (cluster CA, API server, etcd, front proxy, service account keys), bootstraps etcd and the static control-plane pods, and writes the admin kubeconfig (`Kube.config`) at `/etc/kubernetes/admin.conf`. The install script then deploys the CNI plugin, copies the generated certificates and the admin kubeconfig to `/share/pki/` and `/share/Kube.config`, and finally creates a `/share/leader_ready` file as a synchronization barrier.

Follower path - Non-leader control-plane nodes poll `/share` for `leader_ready` every five seconds. Once it appears, they copy the cluster CA and other certificates from `/share/pki/` into `/etc/kubernetes/pki/`, start the Phylactery service and Keepalived, and execute `kubeadm join --discovery-file /root/.kube/config --control-plane` to register as additional control-plane members. As each new follower joins, every Phylactery instance refreshes its HAProxy backend list (Section 3.5.4), so the load balancer adopts the new API server immediately.

Worker path - Worker nodes also poll for `leader_ready`, but they need no PKI certificate of their own, since they host no control-plane component. They read `/share/Kube.config` and execute `kubeadm join --discovery-file /root/.kube/config` to register as workers. The kubelet creates the Node object, the scheduler can place pods on it, and the workload starts.

Single-node and incremental growth - The same machinery supports starting with a single control-plane node and adding more later. Late-joining nodes do not wait for `leader_ready`, which already exists, and proceed directly to the credential fetch and `kubeadm join` steps. This lets the negotiation service add nodes at runtime, not only at cluster creation.

3.5.4 Phylactery and node rejoin

Phylactery solves the two coordination problems the static system overlay cannot address on its own: keeping the HA control-plane endpoint current as members come and go, and reattaching a rebooted control-plane node that has lost all local state.

HAProxy maintenance - Whenever a control-plane node starts, its Phylactery instance writes a small file with its own hostname and IP into `/share`, reads any matching files left by other control-plane nodes, rebuilds the full backend list, and rewrites the local HAProxy configuration. It then exposes a small HTTP endpoint on port 9999 and notifies the Phylactery instances on all other control-plane nodes (through `GET` requests to that endpoint) so they refresh their own HAProxy in turn. Together with Keepalived owning the VIP, the addition or removal of a control-plane node therefore propagates to every load balancer in the cluster within a few round-trip times.

Stateless rejoin - When Warewulf reboots a control-plane node, all local state is lost; the node comes back up with the same hostname and IP and the same install script. Both Kubernetes and etcd refuse the rejoin, because they still hold an entry under that name. Phylactery performs the cleanup. Before the install script reaches `kubeadm join`, the local Phylactery instance reads the admin kubeconfig from `/share`, drains and deletes the stale Node object through `kubectl`, and removes the matching stale etcd member through `etcdctl`. Only then does it signal readiness and allow `kubeadm join` to proceed. Worker rejoin is simpler: The install script itself performs the same `kubectl` cleanup before joining, without Phylactery on the node. Decker and Kunkel point out that this cleanup is why the original architecture stores `Kube.config` on `/share`: Without admin credentials reachable from a freshly rebooted node, Phylactery cannot remove the stale entry and the rejoin would deadlock. Section 3.5.6 comes back to this point.

3.5.5 PKI certificates and credentials

The cluster's identity rests on a small set of certificate pairs and one administrator file:

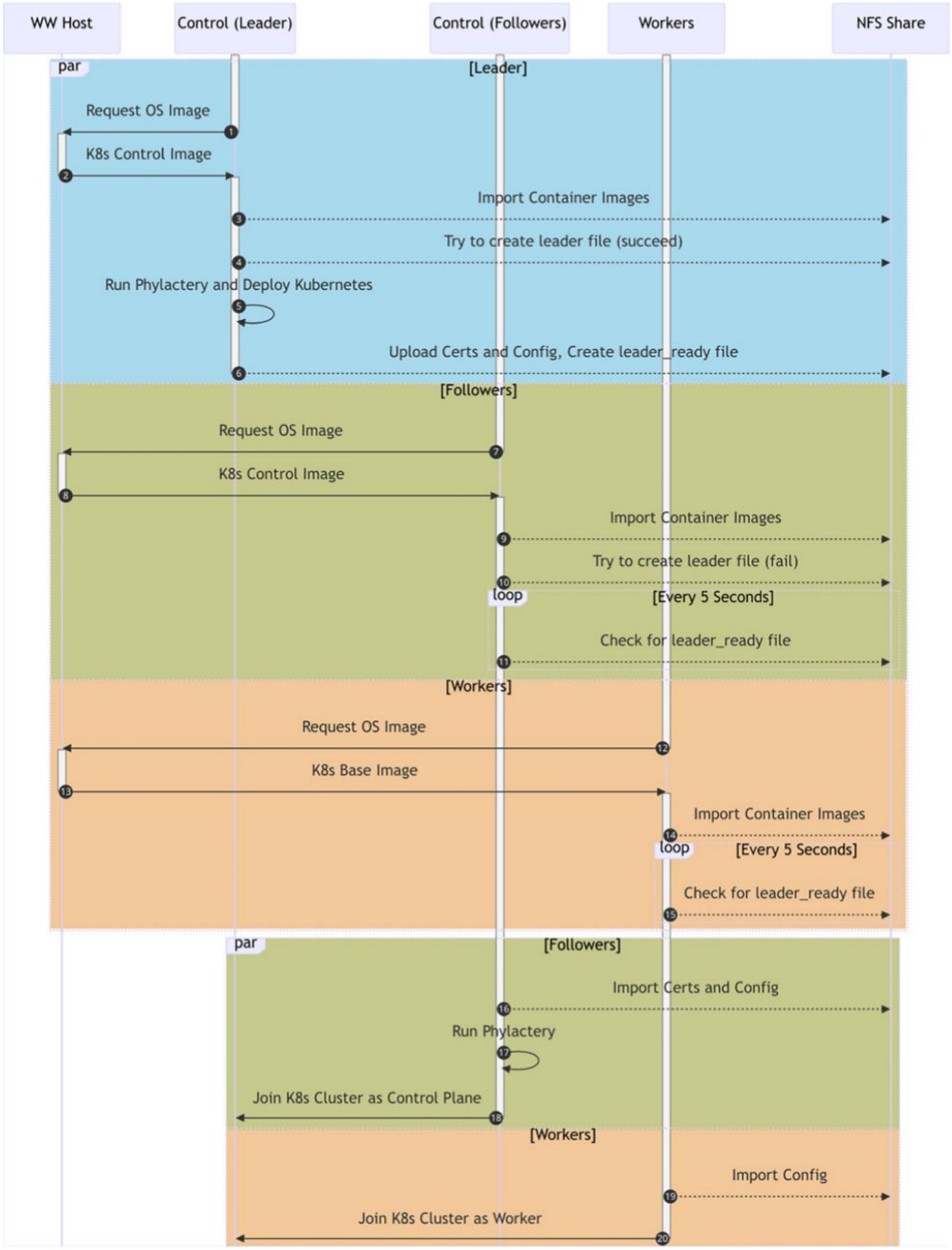


Figure 7: Sequence diagram depicting the deployment process of Ephemeral Kubernetes. The node labeled as Control (Leader) is the same as the Control (Followers) nodes except that it was the first to create the leader file on the NFS share. Taken from the original Ephemeral Kubernetes paper [DK25].

- **Cluster CA** (`ca.crt`, `ca.key`): the master root of trust. Every component certificate in the cluster is signed by this CA, and anyone who has `ca.key` can mint new admin credentials.
- **Service account keys** (`sa.key`, `sa.pub`): sign and verify the bearer tokens mounted into pods. Every control-plane node has to hold the same key pair, so that a token issued by one API server validates on any other.
- **Front-proxy CA** (`front-proxy-ca.crt`, `front-proxy-ca.key`): serves the API aggregation layer, so that extension API servers can authenticate to the main API server.
- **etcd CA** (`etcd/ca.crt`, `etcd/ca.key`): protects communication between etcd members. New control-plane nodes need the etcd CA key to mint their own etcd peer certificates when they join.
- **Admin kubeconfig** (`Kube.config`): bundles the cluster CA certificate, an admin client certificate signed under `Common Name = kubernetes-admin` and `Organization = system:masters`, and the matching private key. Anyone holding this file obtains full control over the cluster.

In the original architecture, `kubeadm init` on the leader node generates all of these, and the install script copies them to `/share/pki/` and `/share/Kube.config`, where they remain on the NFS share so that follower control planes and rebooted nodes can fetch them.

3.5.6 Security analysis and proposed enhancements

The companion report [Mad26] audits this credential model and concludes that it prioritizes deployment speed and operational simplicity over security. Storing `Kube.config` and the raw CA private keys on a shared NFS mount, as sketched in Figure 5, creates three concrete risks.

Privilege escalation - Any user or process that can read `/share/Kube.config` obtains full cluster-admin privileges. With those, they can deploy a privileged container that mounts `/etc/shadow` on its host, replace the root password, and gain root on any node in the cluster [Mad26].

No revocation - Once `Kube.config` or `ca.key` has been copied off `/share`, no mechanism invalidates the leaked credential short of rebuilding the cluster from scratch. The signed admin client certificate inside `Kube.config` is valid until its expiry date, and the cluster CA cannot be rotated in place without reissuing every component certificate.

Lateral movement - Because the same credentials are reachable from every node, the compromise of any single node, worker or control plane, hands an attacker the keys to the entire infrastructure. No per-node identity exists to which a workload-bearing node could be bound.

The report proposes two architectural enhancements, applicable separately or in combination.

Key-based credential distribution - The first enhancement removes sensitive material from the shared `/share` directory entirely and replaces it with a secure HTTPS file server on the Warewulf cluster manager. Credentials live in a private directory (`/var/lib/warewulf/private-secrets`, mode 700, root only) and reach only clients that present a valid per-node key. A pre-deployment script (`generate-Keys.sh`) creates a unique SHA-256 key for each node registered in Warewulf and writes it as a JSON record under `/var/lib/warewulf/Keys/`. The keys are then delivered to the nodes through the Warewulf system overlay, alongside the rest of the `k8s-overlay`, so each node receives exactly its own key as `/etc/k8s-key.{NODE_NAME}` at provisioning time. After `kubeadm init`, the leader uploads the admin kubeconfig and a tarball of the CA certificates and keys to the secure server through authenticated HTTPS POST requests. Followers and workers later download them by presenting their own key. They delete the key from local storage as soon as the cluster join succeeds. Together these mechanisms provide four properties:

- Sensitive material is no longer permanently accessible on a network filesystem.
- Each node authenticates with a unique, auditable identity.
- The attack window for key theft on a node is bounded by the time between boot and successful join, usually under 90 seconds.
- A compromised worker can no longer read cluster-admin credentials, because its key is gone.

Token-based access control - The second enhancement applies the principle of least privilege to whatever is left on `/share`. Instead of the full `Kube.config`, the leader writes only three narrowly scoped credentials. A `join-command.txt` file holds the `kubeadm join` command together with a short-lived bootstrap token permitted to do nothing except join the cluster: Even if it leaks, it cannot read secrets, query workloads, or alter cluster state. A `phylactery-token.txt` file holds a service-account token whose RBAC permissions are restricted to listing, draining, and deleting Node objects, the minimum Phylactery needs for stale-node cleanup. A `certificate-key.txt` file holds the decryption key for the encrypted certificate bundle that `kubeadm init --upload-certs` pushes into the cluster, with which control-plane followers fetch their certificates from the API server itself instead of from NFS. With these three files in place, raw private keys such as `ca.key` are never directly readable on `/share`, and an attacker who breaches the share must still address the API server actively, and generate audit logs, to obtain anything administratively useful.

Residual limitation - Both enhancements still require `ca.key` on every active control-plane node, because Kubernetes needs it locally to sign certificates for new joiners. An attacker who gains root on a running control-plane node can read `ca.key` from local disk and mint arbitrary admin credentials. This risk is inherent in the Kubernetes PKI architecture and cannot be removed by changing how credentials are distributed; protection of the running control plane therefore falls back on the underlying network isolation and node hardening of the HPC site [Mad26].

Implication for the negotiation service - The negotiation service in this thesis treats the secured variant of Ephemeral Kubernetes as the deployment target. Per-node Warewulf overlays carry per-node keys, the Warewulf host runs the secure credential server, and a node in the Slurm role never receives Kubernetes credentials. Every Kubernetes-bound transition issues the node a fresh key through its overlay, and the node deletes the key once `kubeadm join` succeeds.

3.5.7 Gap

Ephemeral Kubernetes is the foundation this thesis builds on rather than a competitor. It already runs a real Kubernetes cluster on the same physical machines Slurm normally manages, at native performance, with no virtualization layer between Kubernetes and the hardware. What it does not address is the dynamic side. In its current form, the choice of Kubernetes or Slurm for a given node is a fixed assignment the administrator sets through the node's Warewulf image; idle Kubernetes capacity does not release itself for Slurm jobs, and pending Slurm jobs cannot prompt Kubernetes nodes to leave the cluster. The negotiation service is the missing control loop: It watches both schedulers, decides when a node should change role, and drives the drain-and-reboot path in both directions, reusing Ephemeral Kubernetes for every join.

3.6 SAIA: Slurm-native hosting for HPC-based services

Doosthosseini et al. [Doo+26; Doo+25] address the service side of the same problem. Their setup is a real production deployment at GWDG: Chat AI, a web service that serves LLM inference to over 100 000 academic users on top of the institution's HPC hardware. The constraint they accept is that the HPC site's security policies (no root on compute nodes, no direct network access to compute nodes, SSH-only access through login nodes) cannot be loosened, so a Kubernetes deployment is not an option. SAIA is built entirely on Slurm. A scheduler component decides how many service instances to keep alive and submits them as ordinary Slurm jobs; a proxy on the login node accepts the external HTTPS traffic and forwards requests through confined SSH commands to whichever compute nodes Slurm has placed the service instances on. An autoscaling component observes per-instance demand and adjusts the number of running Slurm jobs accordingly.

Gap - SAIA shows that real-time LLM inference can be hosted on a Slurm cluster under strict security rules, and it does so on the same GWDG hardware this thesis targets. Because every service instance is a Slurm job, the platform inherits Slurm's batch scheduling behavior: The queue wait of the job determines startup latency, scaling decisions await the scheduling of new jobs, and features Kubernetes provides natively (rolling updates, readiness probes, service meshes, autoscaling on metrics other than queue depth) have to be rebuilt on top of Slurm primitives. This thesis explores the option SAIA set aside: Host the service layer on a real Kubernetes cluster, accept the administrative cost of operating one, and return hardware to Slurm batch jobs whenever the inference workload does not require it.

3.7 Inference scaling with a Kubernetes front end

Trappen et al. [Tra+25] build a system close in spirit to SAIA, but with Kubernetes added on the front-end. Their goal is the same: serve LLM inference to higher-education users on an HPC system, in their case the RAMSES supercomputer. They divide the system into two layers. The first layer is a Kubernetes-based web API: a web gateway that authenticates requests and forwards them, a PostgreSQL database, and a set of worker services that manage the models. The second layer is the models themselves, which execute vLLM inside Slurm jobs, started through Apptainer. A job worker submits or removes vLLM Slurm jobs, and the number of model instances follows vLLM's own load metrics: In their tests, a vLLM queue time above five seconds for 30 seconds started

another instance. Prometheus and Grafana make these metrics visible. Their benchmark shows the architecture serves 100, 500, and 1000 concurrent requests with about 500 ms of extra end-to-end latency, and that the web gateway becomes the bottleneck once enough GPU compute is present.

Gap - Trappen et al. apply Kubernetes only to the front-end services; the inference itself still executes as Slurm jobs, so the compute nodes always belong to Slurm. They scale inference by submitting more vLLM Slurm jobs rather than by moving compute nodes between the two systems. They therefore do not balance HPC compute power between on-demand Kubernetes services and batch Slurm jobs: No Kubernetes worker pool grows and shrinks against a Slurm batch pool, and no node-sharing mechanism returns idle inference capacity to batch users. The model serving also inherits Slurm's batch behavior, the same limit as SAIA (Section 3.6): A new model instance has to be submitted as a Slurm job and wait in the queue, and the job's wall-time limit bounds it, instead of persisting as a long-lived Kubernetes service on its own nodes. The negotiation service in this thesis takes the other path: It hosts the inference service on a real Kubernetes cluster with its own worker nodes and moves those nodes between Kubernetes and Slurm as demand shifts, so that the same hardware serves on-demand inference and batch jobs without either side being fixed in advance.

3.8 Workload forecasting and Chronos-Bolt

The last strand of earlier work concerns forecasting. The predictive policy of the negotiation service needs one building block that none of the systems above provides: a forecast of the inference load of the coming hour. This thesis takes that forecast from Chronos-Bolt, a pretrained time series model from the Chronos family developed at Amazon [Ans+24]. This section first outlines the main families of workload forecasting methods, then explains the idea behind Chronos-Bolt, the data it was pretrained on, and why this thesis prefers it to the alternatives; Section 4.2.5 describes how its forecast becomes a node count.

Other ways to forecast a workload - Workload forecasting is an old problem, and most earlier methods fall into three families. The first family fits a small statistical model to the history of a single series. The classic example is ARIMA, which predicts the next value as a weighted sum of the last few observed values and the last few forecast errors; Calheiros et al. applied it to predict web request rates and provision cloud servers before the demand arrived [Cal+15]. These models are simple and well understood, but each series needs its own fit, and that fit decays once the pattern changes. The second family trains a neural network on the history of the workload itself. Kumar and Singh trained such a network for cloud data centers [KS18], and later work such as esDNN relies on recurrent networks, which read the series step by step and carry a memory of what preceded [Xu+22]. These networks capture patterns ARIMA misses, but they need a long history for training and retraining as the workload drifts. The third family avoids modeling altogether. Shahrad et al. maintain, for every serverless function on Azure, a small histogram of the gaps between its calls and warm the function shortly before the next call is likely [Sha+20]. This costs almost nothing at the scale of millions of functions, but it predicts when the next event arrives, not how high the load will be.

Why a pretrained model - Closest to this thesis, measurements of production LLM services show request streams with a strong daily rhythm and short bursts on top, both in a months-long trace of Azure GPT services [Wan+25] and across Alibaba's serving

platform [Xia+26]. These studies do not forecast the load, but they confirm that a load like SAIA's is regular enough for forecasting to work. The newest family of methods makes such forecasting cheap. Models such as Chronos [Ans+24] and Google's TimesFM [Das+24] are trained once, by their authors, on huge collections of series from many domains, and can then forecast a series they have never seen without any training on it. For the negotiation service this removes the main cost of the first two families: No per-series fit requires maintenance and no training pipeline requires operation. The rest of this subsection explains the member of this family the thesis adopts.

Forecasting as language modeling - A language model reads a sequence of words and predicts the word that comes next. Chronos [Ans+24] applies the same recipe to time series. A series of measurements, such as the number of busy GPUs in each hour, is scaled to a common range and rounded into a fixed set of bins, so that every observation becomes a token from a fixed vocabulary, the way a word is a token in a sentence. A network from the T5 language model family is then trained to predict the next token from the tokens before it, with the same training objective used for text; nothing about the architecture is specific to time series. Because the model assigns a probability to every possible next token, it does not return a single guess but a spread of possible futures, which can be read out as quantiles: a middle estimate with a band around it.

What the model was trained on - The authors trained Chronos once, on a large collection of publicly available time series from domains that have nothing to do with each other: energy demand, road traffic, weather, retail sales, web requests, and more, extended with randomly generated series for variety [Ans+24]. The point of this pretraining is transfer. Daily cycles, weekly cycles, trends, and noise look much the same across domains, so a model that has seen millions of series can forecast one it has never seen. Applying the model in this way, with downloaded weights and no further training, is called *zero-shot* forecasting, and it is how this thesis proceeds: The model never sees GWDG data until the moment it is asked to forecast it.

Chronos-Bolt - Chronos-Bolt is the faster second generation of the family, released by the same group in late 2024.[7] The original Chronos predicts one token at a time, so a multi-hour forecast needs one model call per step. Chronos-Bolt instead groups the input history into patches of several observations and produces the whole forecast horizon in a single pass, directly as nine quantiles per future step (the 10th to the 90th percentile). It was trained on almost 100 billion observations, and the authors report it to be up to 250 times faster than original Chronos models of the same size while also being more accurate. The models are small by the standards of modern LLMs, 9 to 205 million parameters. This thesis adopts the largest variant, `chronos-bolt-base`, which returns a forecast in well under a second on an ordinary CPU.

Why this model fits the task - Three properties make Chronos-Bolt a good match for the negotiation service. First, it needs no training: The weights are downloaded once and applied as delivered, so the forecasting step brings no training pipeline and no model-maintenance burden of its own. Second, it returns a band rather than a single number, and the predictive policy uses the band directly: The service provisions the inference side to the upper edge of the band, because guessing too high costs a few lendable nodes while guessing too low costs inference capacity. Third, it is cheap and workload-agnostic: One

[7] Chronos-Bolt model card, Hugging Face: `https://huggingface.co/amazon/chronos-bolt-base`, accessed 30 June 2026.

forecast per hour costs well under a second of CPU time on the control host, and the model sees only a sequence of numbers, so nothing in this step is tied to LLM inference in particular. The earlier families offer none of this without additional effort: An ARIMA fit or a trained network is tied to the one series it learned, requires its own training and retraining pipeline beside the negotiation service, and would have to learn from a single month of SAIA history, which is thin training material for a neural network. Within the pretrained family, this thesis selects Chronos-Bolt because its weights are open and freely downloadable, it ships as a small Python library that returns the quantile band directly, and it is fast enough on an ordinary CPU that the control host needs no GPU of its own.

Gap - Time-series forecasting is a mature field with strong, simple baselines [HA21], and pretrained models such as Chronos [Ans+24] now produce usable forecasts, with uncertainty ranges, on series they were never trained on. Most of that literature measures how accurate a forecast is. The earlier systems that do act on a forecast, such as the predictive algorithm of the Balancer [Liu+18] and the ARIMA-based provisioning of Calheiros et al. [Cal+15], fit a model to each workload's own history. What remains untested is the step this thesis needs: feeding a pretrained, zero-shot forecast into a capacity decision between two GPU clusters. This thesis tests that step for one case: Section 4.2.5 converts the forecast's upper band into a node count, and the evaluation judges the forecast by the queue wait it saves and the requests it delays, not only by its error metrics.

3.9 Summary and positioning

Table 1 assesses the seven systems described above against the four requirements the negotiation service has to meet. The requirements are (i) *dynamic node reassignment*, in which the boundary between Kubernetes and Slurm hardware moves at runtime in response to load; (ii) *bare-metal performance*, in which the service workload executes without an additional virtualization or container-translation layer between the application and the GPU; (iii) *Kubernetes-native services*, in which Kubernetes service abstractions (deployments, services, ingresses, HPA) function directly rather than being rebuilt on top of another scheduler; and (iv) *Slurm compatibility*, in which existing Slurm users continue to submit jobs through `sbatch` and `srun` unchanged.

Read row by row, the table shows the gap. Liu et al. share the negotiation idea but target VM-based on-demand clusters rather than bare-metal Kubernetes; the Torque-Operator addresses a different problem, cross-scheduler job submission, and rests on a scheduler the target site does not operate; HPK and KSI provide Kubernetes semantics on HPC hardware, but only inside a Slurm allocation, which forfeits bare-metal performance and binds the Kubernetes cluster's lifetime to a batch job; Ephemeral Kubernetes provides the bare-metal Kubernetes foundation the thesis needs, but moves no node in response to demand; SAIA demonstrates that real-time LLM inference works on the GWDG cluster, but only by hosting the service layer inside Slurm; Trappen et al. place a Kubernetes front-end on a Slurm-hosted inference service and scale it by submitting more Slurm jobs, but never move compute nodes between the two schedulers. The contribution of this thesis is the row at the bottom: a Warewulf-hosted negotiation service that exploits Ephemeral Kubernetes' image-based provisioning to move nodes between a real Kubernetes cluster, which hosts the LLM inference service natively, and Slurm, which continues to execute batch jobs unchanged, with the boundary driven by observed demand rather than by fixed administrative assignment.

Table 1: Comparison of related systems against the four requirements of the dynamic node negotiation service.

System	Dynamic node reassignment	Bare-metal performance	Kubernetes-native services	Slurm compatibility
Balancer (Liu et al.) [Liu+18]	yes	VMs on the on-demand side	no (OpenStack)	no (Torque)
Torque-Operator [Zho+21]	no	under Singularity	partial	no (Torque)
HPK [Cha+23]	no	under Apptainer	yes (via Slurm)	yes
KSI [DMK25]	no	under Kind/ Podman	yes (over Slurm)	yes
Ephemeral Kubernetes [DK25]	no (static role)	yes	yes	possible (separate image)
SAIA [Doo+26]	no	yes	no	yes
Trappen et al. [Tra+25]	no	under Apptainer	front-end only	yes
This thesis	**yes**	**yes**	**yes**	**yes**

4 Methodology

This chapter translates each research question into a concrete design problem (Section 4.1), presents the design of the negotiation service and its reference implementation (Section 4.2), defines the metrics that measure success (Section 4.3), and describes the experiments that produce them (Section 4.4).

4.1 Research questions as design problems

The central question of Section 1.2 divides into four narrower ones, each of which poses a design problem.

RQ1: Can a Warewulf-provisioned cluster switch the same physical node between a Kubernetes role and a Slurm role? - The design problem is to build a Warewulf setup in which the same physical node boots as a Kubernetes worker one time and as a Slurm compute node the next, with the role determined entirely by the node's image assignment on the Warewulf host. A role change must require no manual step on the node itself.

RQ2: Can the negotiation service switch a node's role automatically, without disrupting running work? - The design problem is for a daemon on the Warewulf host to detect that one side needs more nodes, wait for any work on the chosen node to finish, change the node's image assignment in Warewulf, and reboot the node into its new role, all without human intervention and without killing any job or request already in progress.

RQ3: Does automatic node sharing reduce idle node time and Slurm queue wait compared to a static split, and does predicting the inference load help beyond reacting to it? - The design problem is to define a policy that decides when to move a node and which node to select, so that idle node time and Slurm queue wait fall without degrading the Kubernetes side noticeably. Section 4.2.5 proposes and describes two policies: an *on-demand* rule that moves a node only when one side needs it at that moment, and a *predictive* policy that forecasts the inference load of the coming hour and moves nodes before the need arises. The question is whether either improves on the static baseline of Figure 12, and how close each comes to an *oracle* that knows the future load exactly.

RQ4: What is the cost of one transition, and which cooldown period stops the system from oscillating? - The design problem is twofold. First, measure how long a single transition takes, from Kubernetes to Slurm and from Slurm to Kubernetes. Second, choose how long a node must remain idle in Slurm before the service returns it to Kubernetes, the cooldown (`IDLE_THRESHOLD`), and check that this cooldown keeps the system from moving a node back and forth faster than either side can use it.

4.2 Design of the negotiation service

The proposed solution is the *negotiation service*: a single daemon on the Warewulf host that moves compute nodes between two clusters on the same hardware, a Kubernetes cluster that hosts service workloads and a Slurm cluster that executes batch jobs. The design adds no software to the compute nodes. Section 3 showed why none of the existing

systems achieve this in a form that suits GWDG; this section presents the new design. The source of the reference implementation is public, and it builds on the Ephemeral Kubernetes scripts [DK25]; Table 7 gives both repositories together with the exact commits behind the results of this thesis.

The section is organized as follows. Section 4.2.1 states what the system has to do and what it must not do. Section 4.2.2 describes the Warewulf base: the two boot images and the division between fixed nodes and the dynamic pool. Section 4.2.3 introduces the negotiation service: where it runs, what it reads, and what it commands. Section 4.2.4 presents a single node transition step by step, in both directions. Section 4.2.5 describes the two policies for deciding when to move a node: an *on-demand* Kubernetes-first rule that reacts to observed state, and a *predictive* policy that forecasts the inference load and acts ahead of it. Section 4.2.6 gives the reasons behind the main design choices, and Section 4.2.7 presents the reference implementation.

4.2.1 Requirements

The four requirements from Section 3.9 carry over: Nodes have to move between clusters at runtime, the moved nodes have to retain bare-metal performance, the Kubernetes side has to appear to its users as ordinary Kubernetes, and the Slurm side has to appear to its users as ordinary Slurm. Five further rules follow from the GWDG setup and from the Ephemeral Kubernetes work this thesis builds on.

No state persists across a role change - As described in Section 2.1.4, compute nodes at GWDG have no local disk that survives a reboot. The root filesystem is in RAM and is wiped on every boot. The new design adopts this as a strict rule: Nothing left on the node from a previous role may leak into the next role. The next boot defines what the node is.

No additional daemon on a compute node - Each role already places its own stack on a compute node: `slurmd` and `munged` in the Slurm role, the kubelet and a container runtime in the Kubernetes role. One additional daemon on every node purely to coordinate moves would add code to maintain, further failure modes, and further attack surface. The entire movement logic therefore runs on the management side, not on the compute nodes.

Fixed nodes remain untouched - The Kubernetes control plane (`kube-apiserver`, `etcd`, scheduler, controller-manager) and the Slurm controller (`slurmctld`) carry the state of both clusters. The negotiation service may not touch them. Only nodes in the dynamic pool change role.

Clean accounting on both sides - Compute time at GWDG is billed per account (Section 2.1.2). When a node is on the Kubernetes side, no Slurm user may be billed for it, and when it is on the Slurm side, Kubernetes must not count it as a worker. Every move therefore has to remove the node cleanly from the side it leaves, not only add it to the side it enters.

Secured Ephemeral Kubernetes as the deployment target - The system builds on the Ephemeral Kubernetes framework [DK25] and on the secured variant from the companion report [Mad26]. Per-node keys reach a node only through its Warewulf overlay; the Warewulf host operates the secure file server that issues the cluster credentials; a node in the Slurm role never holds Kubernetes credentials. The negotiation service reuses this machinery on every move.

4.2.2 The Warewulf base

The whole design rests on one property of Warewulf, presented in Section 2.2: A node is defined by the pairing of a Warewulf *image* (a read-only base operating system) with one or more *overlays* (per-node configuration files). Rebooting a node reapplies that pairing. To switch a node from Kubernetes to Slurm, the design changes the image and overlay assignment on the Warewulf host and then reboots the node. Everything else follows from that.

All overlays in this design are *system overlays* in the sense of Section 2.2: They are built into the boot image and applied once, before `systemd` starts. The service never refreshes a node's configuration through a runtime overlay. Every change reaches a node by an edit to the system overlay on the Warewulf host, a rebuild with `wwctl overlay build`, and a reboot. The design uses no runtime overlays on purpose: Every role change already includes a reboot (Section 4.2.4), so a second, in-place update path would add complexity without any benefit.

Images and overlays of the two roles - The Warewulf host carries four images and three overlays, combined into two overlay sets; all three overlays are system overlays. The images are `k8s-control-ww` (for Kubernetes control-plane nodes), `k8s-worker-ww` (a copy of the control image kept under its own name so worker-side changes can land without touching the control image), `slurm-ctrl-ww` (for the Slurm controller), and `slurm-compute-ww` (for Slurm compute nodes). The two overlay sets are `wwinit,k8s-overlay` (basic per-node setup plus the Ephemeral Kubernetes install script, the Phylactery service, and the file that delivers the per-node Kubernetes key) and `wwinit,slurm-overlay` (basic per-node setup plus `slurm.conf`, `cgroup.conf`, the Munge key, and the boot-time `slurm-install.sh` script that starts `slurmd`). `wwinit` is a system overlay because the files it carries (hostname, network configuration, SSH host keys) are needed before networking comes up; `k8s-overlay` and `slurm-overlay` are system overlays because they ship the `systemd` unit files and oneshot drivers that have to be present in the filesystem the moment `systemd` starts as PID 1. The `wwinit` overlay is shared because it only carries generic items (hostname, network, SSH keys). The file layout of all three overlays is sketched in Figure 8.

Fixed nodes and the dynamic pool - Each physical node falls into one of three groups, written into a single file `node-config.conf` on the Warewulf host. *Fixed Kubernetes nodes* (`cluster=k8s`) hold the control plane and any permanent Kubernetes workers; the service never moves them. *Fixed Slurm nodes* (`cluster=slurm`) hold `slurmctld` and any permanent Slurm compute nodes; the service never moves them either. *Dynamic nodes* (`cluster=dynamic`) are the pool the service rebalances. Every line in the file lists the hostname, MAC address, IP address, cluster, and role; for dynamic nodes there is also a `current_cluster` field that says where the node is right now (`k8s`, `slurm`, or `none` for a node that has not been assigned yet). Figure 9 shows a small example. Reading and writing this file is how the service remembers its decisions across restarts.

Pre-declaration of every dynamic node in Slurm - The design declares every node of the dynamic pool inside Slurm's `slurm.conf` (under `NodeName=`) at provisioning time, whether the node currently carries a running `slurmd` or not. When the service moves a node into Slurm, `slurmctld` already knows it; when it moves the node out, the entry simply turns `DOWN`. Without this, every move would need an edit to `slurm.conf` followed by `scontrol reconfigure`. Pre-declaring the nodes avoids that step and ensures that

```
/var/lib/warewulf/overlays/
+- wwinit/                                (shared; applied in both roles)
|  +- etc/hostname.ww                     hostname, templated per node
|  +- etc/sysconfig/network-scripts/      IP, gateway, DNS
|  \- root/.ssh/authorized_keys.ww        admin SSH key
|
+- k8s-overlay/                           (applied only in Kubernetes role)
|  +- k8s-helper/
|  |  +- k8s-install.sh                   oneshot kubeadm init/join driver
|  |  \- phylactery.py                    Phylactery sidecar service
|  +- etc/systemd/system/
|  |  +- k8s-install.service              runs k8s-install.sh at boot
|  |  \- phylactery.service               runs phylactery.py at boot
|  \- etc/k8s-key.<hostname>              per-node SHA-256 key (deleted after join)
|
\- slurm-overlay/                         (applied only in Slurm role)
   +- etc/slurm/
   |  +- slurm.conf                       cluster-wide Slurm config
   |  \- cgroup.conf                      per-job cgroup limits
   +- etc/munge/munge.key                 shared Munge authentication key
   +- root/slurm-install.sh               oneshot slurmd starter
   \- etc/systemd/system/
      \- slurm-install.service            runs slurm-install.sh at boot
```

Figure 8: Contents of the three Warewulf overlays on the Warewulf host. All three are *system overlays*: Their files are baked into the boot image and applied before `systemd` starts. `wwinit` is shared and carries only generic per-node items (hostname, network, SSH key). `k8s-overlay` adds the Ephemeral Kubernetes install driver, the Phylactery sidecar service, and the per-node Kubernetes key that authorizes the node to fetch its credentials from the secure server in Section 3.5.6. `slurm-overlay` adds the Slurm and cgroup configuration, the Munge authentication key, and a oneshot service that starts `slurmd` at boot. Warewulf applies `wwinit,k8s-overlay` to nodes assigned to the Kubernetes role and `wwinit,slurm-overlay` to nodes assigned to the Slurm role, so a node never holds both sets at the same time.

the controller and the negotiation service always agree on which nodes exist. Kubernetes needs no equivalent step: A worker is in the cluster only while its Node object exists, and `kubeadm join` creates that object on every entry.

Credential separation between the two roles - The Warewulf host operates the secure credential server from Section 3.5.6: an HTTPS Flask service on port 8000 that issues the admin kubeconfig and the PKI bundle only to a node presenting a valid per-node SHA-256 key. The key reaches the node through its Kubernetes overlay as `/etc/k8s-key.`*`hostname`*, and the node deletes it once the join succeeds. When the same node later holds the Slurm role, its image is `slurm-compute-ww` and its overlay is `wwinit,slurm-overlay`, so it holds no Kubernetes key and no Kubernetes credential at any time. This is the property the trust model in Section 3.5.6 required.

```
# Fixed Kubernetes nodes (never reassigned)
# HOSTNAME       MAC                IP          CLUSTER  ROLE
control0         fa:16:3e:9d:d4:f1  10.0.0.2    k8s      k8s-control
control1         fa:16:3e:75:3e:d8  10.0.0.16   k8s      k8s-control

# Fixed Slurm nodes (never reassigned)
# HOSTNAME       MAC                IP          CLUSTER  ROLE
slurm-control    fa:16:3e:2c:47:17  10.0.0.14   slurm    slurm-controller
slurm-compute01  fa:16:3e:80:7b:76  10.0.0.12   slurm    slurm-compute

# Dynamic pool (rebalanced by the negotiation service)
# HOSTNAME       MAC                IP          CLUSTER  ROLE     DYNAMIC_STATE
dynamic-node     fa:16:3e:e1:54:61  10.0.0.25   dynamic  dynamic  current_cluster=none
```

Figure 9: Representative `node-config.conf` for a small hybrid deployment. The first block lists fixed Kubernetes nodes (here two control-plane nodes), the second block lists fixed Slurm nodes (one controller and one permanent compute node), and the third block lists the dynamic pool (one node, currently unassigned). The `current_cluster` field on a dynamic line is the only one the negotiation service ever rewrites; the rest are set once at provisioning time.

4.2.3 The negotiation service

The negotiation service is one long-running program on the Warewulf host. It is the only new component the design adds, and the only component that decides when a node changes role. This section describes where it runs, what it observes, and what it commands; Figure 10 shows it in place, as the counterpart of the static baseline in Figure 12.

Location and privileges - The service is a systemd unit (`hybrid-balancer.service`) on the Warewulf manager. It runs as `root`, because every action it performs (invoking `wwctl`, editing overlay files, rebooting nodes) requires root on that host, and because the Warewulf host is already the highest-privilege machine in the cluster: It owns the images, the overlays, the DHCP and TFTP configuration, and the credential server. On that machine the service can read and command both schedulers and keep its state in one local file (`node-config.conf`), without any distributed coordination.

Inputs - The service reads four pieces of state.

- **The list of nodes.** It re-reads `node-config.conf` so that an operator can add or remove a dynamic node by editing the file, without restarting the service.
- **The Slurm side.** It invokes `squeue` and `sinfo` on the Slurm controller over SSH. The number of pending jobs that cannot start on the currently idle Slurm nodes signals that the Slurm side needs more nodes.
- **The Kubernetes side.** It calls the Kubernetes API server (using the admin kubeconfig, or the scoped credential delivered under the secured variant) to read the list of Node objects and the pods running on each one. Empty dynamic nodes show spare capacity on the Kubernetes side, and unscheduled inference pods show that it needs more nodes.
- **Time and history.** It keeps a small in-memory record per dynamic node: when the node last moved, how long it has been idle on its current side, and whether a move

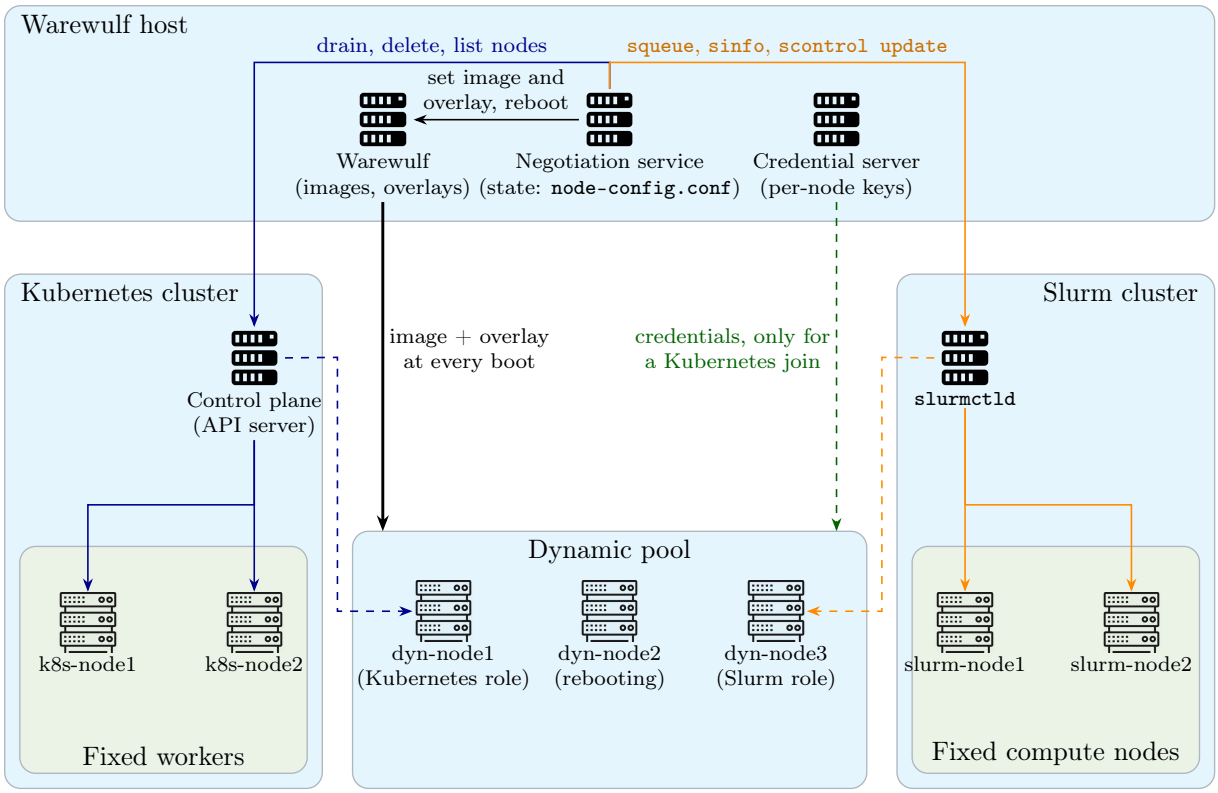


Figure 10: The negotiation service in place. One daemon on the Warewulf host reads both schedulers and drives every move: It drains a node through the Kubernetes API or `scontrol`, sets the node's image and overlay in Warewulf, and reboots it. Warewulf delivers the image and overlay that set a node's role at every boot, and the credential server hands a node its Kubernetes credentials only when it joins that side. The dynamic pool is drawn as a snapshot with one node in each state; the dashed edges show connections that exist only while a dynamic node holds that role. The fixed nodes never move.

is already in flight. It also retains the recent history of inference demand, because the predictive approach of Section 4.2.5 passes that history to its forecasting model.

Decisions - For each dynamic node, the service produces one of four outcomes: *hold* (no action), *move to Slurm*, *move to Kubernetes*, or *reserve for a later move* (the node is busy now, but a move is queued for the moment it falls idle). Which outcome it produces depends on which of the two approaches in Section 4.2.5 is in force at the time.

Actions - When the service decides to move a node, it issues four kinds of action in order.

- **Drain the old side.** For a node leaving Kubernetes, the service invokes `kubectl drain --ignore-daemonsets --delete-emptydir-data` and then `kubectl delete node`. For a node leaving Slurm, it invokes `scontrol update NodeName=`*`n`* `State=DRAIN` on the controller, waits for the current job, if any, to finish, and then sets `State=DOWN`.
- **Reassign in Warewulf.** It invokes `wwctl node set --container` *`image`* `-O` *`overlays`* to switch the image and overlay assignment, followed by `wwctl overlay build` and `wwctl configure dhcp`, so that the change takes effect at the next boot.

- **Reboot.** It invokes `wwctl ssh` *node* `reboot` on the node. (The BMC power-cycle path from Section 2.2 is a drop-in fallback for sites where SSH on the outgoing image is unavailable.)
- **Update the state file.** It writes the new `current_cluster` value back into `node-config.conf`, one line edit per node.

Failure handling - The service attempts no repair of a failed move. If `kubeadm join` or `slurmd` registration does not succeed within a timeout, the service writes a row into its log and CSV file (Section 4.2.7 describes these records) and leaves the node untouched until the next decision. The drain and delete steps ignore their own errors (the Bash idiom `|| true`), so that a partial failure on the side the node leaves does not block the rest of the move; at worst the operator removes a stale entry by hand.

4.2.4 Lifecycle of a single transition

A *move* is the sequence of steps that carries a single dynamic node from one role to the other. The two directions differ on the surface, because Slurm and Kubernetes expose different primitives, but they share the same five phases: *drain*, *reassign*, *reboot*, *join*, *verify*. Figure 11 shows both directions as the sequence of commands the reference implementation issues.

From Kubernetes to Slurm - A Kubernetes-side dynamic node leaves the cluster in this order.

1. **Cordon and drain in Kubernetes.** The service invokes `kubectl drain` *node* `--ignore-daemonsets --delete-emptydir-data --force --timeout=60s`. The node is marked unschedulable, and the inference pods on it are evicted through the eviction endpoint. If the inference Deployment carries a PodDisruptionBudget (Section 2.4.7), each eviction waits until replacement pods run on the remaining Kubernetes nodes, up to the 60-second timeout.
2. **Delete the Node object.** The service invokes `kubectl delete node` *node*, which removes the node from etcd. When the node later returns from Slurm, it therefore joins as a new node.
3. **Reassign in Warewulf.** The service invokes `wwctl node set` *node* `--container slurm-compute-ww -O wwinit,slurm-overlay` and `wwctl overlay build`. The node is already listed in `slurm.conf` (Section 4.2.2), so the Slurm configuration does not change.
4. **Reboot.** The service invokes `wwctl ssh` *node* `reboot`. The outgoing Kubernetes image still runs an SSH daemon, so the node shuts down cleanly.
5. **Verify the Slurm join.** The service polls `sinfo -N` on the controller until the node responds, then invokes `scontrol update NodeName=`*node* `State=IDLE` if the node still shows `DRAIN` from its previous stay in Slurm. The move counts as successful once the controller holds the node in `IDLE` or `ALLOCATED` state.

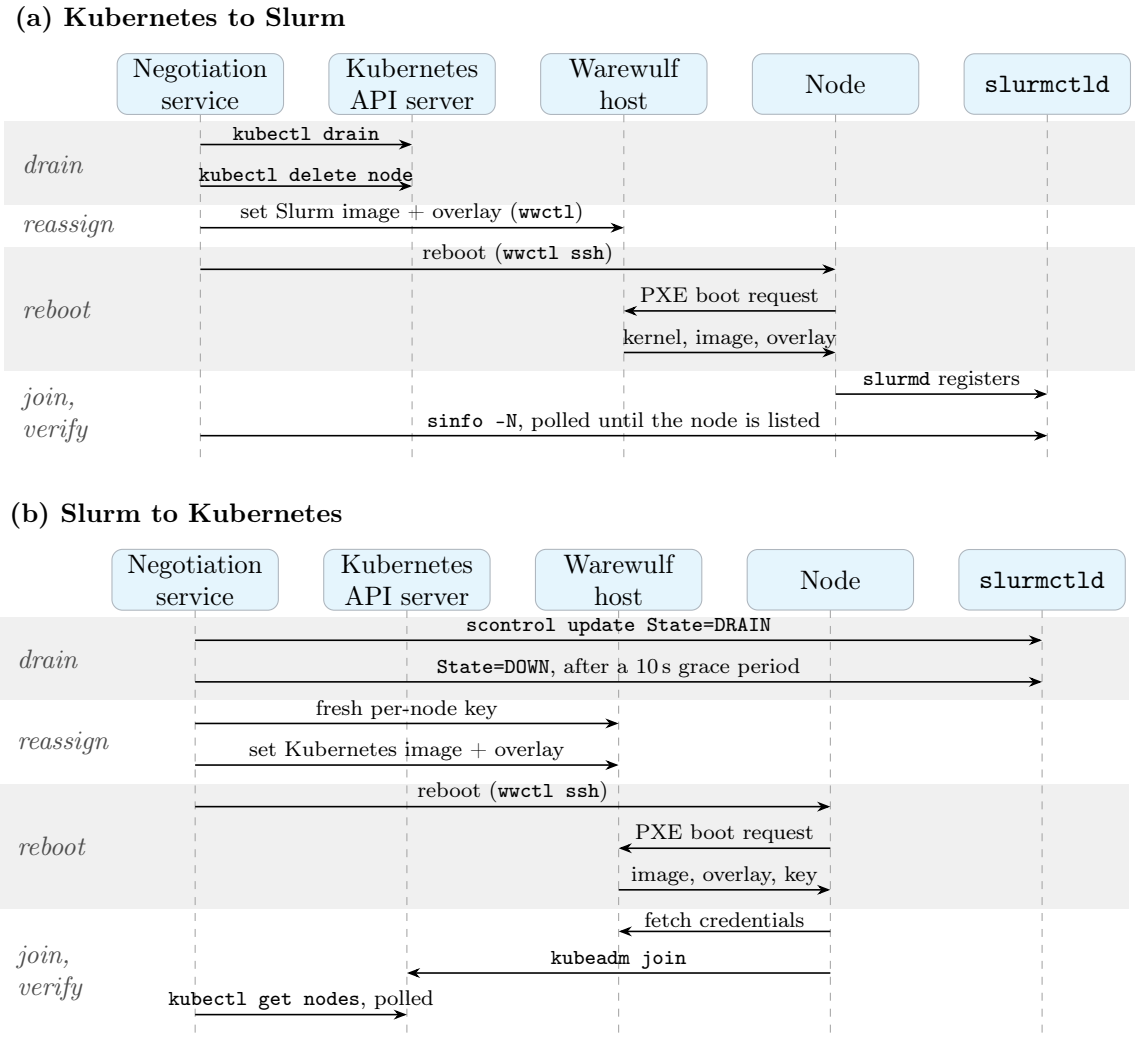


Figure 11: One transition in each direction, as the reference implementation performs it: (a) from Kubernetes to Slurm, (b) back. Time runs downward, and every arrow is a command or a message between two components. The shaded bands mark the phases; join and verify form one band, because the service notices a join only when its check of the new side succeeds.

From Slurm to Kubernetes - The reverse direction uses the same five phases.

1. **Drain in Slurm.** The service invokes `scontrol update NodeName=`*`node`* `State=DRAIN Reason='Moving to K8s'` on the controller. Slurm allows any running job to finish and marks the node `DRAINED`. After a short grace period (10 s in the reference setup), the service invokes `scontrol update State=DOWN` to take the node out of service before the reboot.

2. **Issue the node a fresh Kubernetes key.** The service generates a new SHA-256 key for the node, writes it into the Warewulf token store (`/var/lib/warewulf/tokens/`*`key`*`.json`), and writes it into the Kubernetes overlay as `/etc/k8s-key.`*`node`*. This replaces any older key for the same node. The node carries no Kubernetes credential at rest while it holds the Slurm role, and the fresh key exists only between this step and the join.

3. **Reassign in Warewulf.** The service invokes `wwctl node set` *node* `--container k8s-worker-ww -O wwinit,k8s-overlay` and `wwctl overlay build`. The overlay now carries the fresh per-node key together with the Ephemeral Kubernetes install script and the Phylactery files from Section 3.5.2.

4. **Reboot.** Same as above.

5. **Verify the Kubernetes join.** The service waits for the standard Ephemeral Kubernetes warm-up (NFS mount, container-image import, and `kubeadm join`, as described in Section 3.5.3), then polls `kubectl get nodes` until the node appears with `Status=Ready`.

Time budget of a transition - A single move takes at least as long as a node needs to boot and join the new side. Decker and Kunkel [DK25] report under 90 s for a five-node Kubernetes cluster to form from cold. A single-node move goes through the same image fetch and `systemd` startup, plus a drain on the side it leaves, which depends on the running work and can be the longest part. The design splits the budget in two: *command-to-reboot* time (from the moment the service decides to move to the moment the reboot command leaves the Warewulf host, which is essentially the drain delay), and *reboot-to-join* time (from the reboot to the verified join on the new side). The service writes both values to `boot-times.csv` on every move; Section 5.1 reports a median of 98 s from Kubernetes to Slurm and 102 s back.

4.2.5 Decision approaches

The design separates *how a single node moves* (the five phases in Section 4.2.4) from *when nodes move*. The first part is fixed; the second part is a policy, and the thesis proposes two forms of it: an *on-demand* approach that moves a node when one side runs short at that moment, and a *predictive* approach that forecasts the inference load and moves nodes before the change arrives. Neither approach alters any code inside the five phases; they differ only in when they trigger a move. Without any policy the dynamic pool would remain locked to whichever side each node was first assigned to, with no movement between clusters, as in Figure 12.

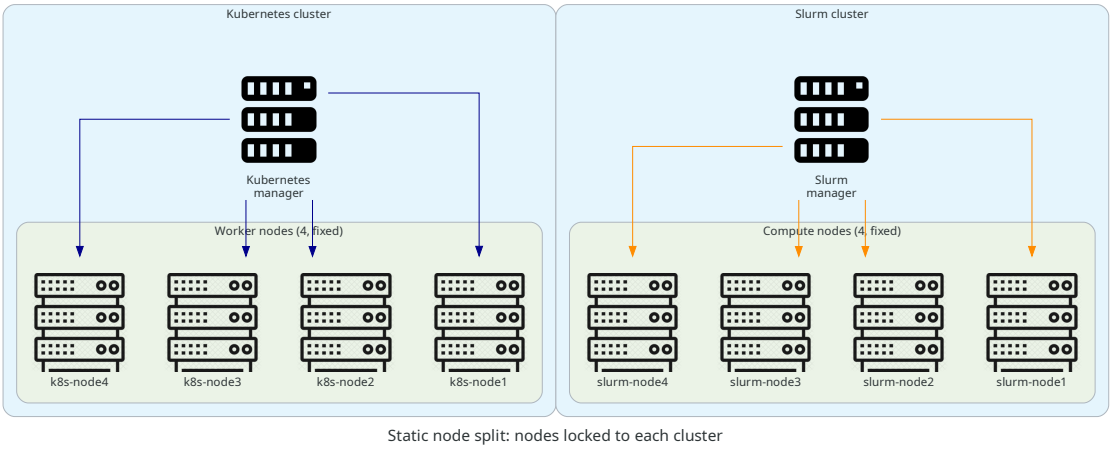


Figure 12: Baseline without a negotiation service: Each cluster is managed on its own and worker nodes stay on the side they were first assigned to.

The two approaches rest on the same observation: The two clusters do not carry the same load at the same time of day. Real GWDG traces show this for both sides (Section 5.2). For Slurm, a six-day `sacct` export from the Grete cluster shows batch GPU jobs arriving across the whole day and a pool that stays busy day and night. For Kubernetes, the month-long export from the SAIA inference platform shows inference to be a service people query mostly in working hours: The busiest hour of the day carries almost sixty times the load of the quietest, the working day is about five times busier than the night, and weekdays are well over twice as busy as weekends. Table 2 puts numbers on this shape for the weekdays of the replayed week. Inference demand follows the clock. At night the service receives fewer than two hundred requests an hour and never needs more than four of its eighteen nodes; in the morning and afternoon it receives over two thousand an hour and at its peak needs all eighteen. Batch demand does not follow the clock. Submissions vary, and they peak in the morning, but the GPUs in use stay close to 300 in every window, because the queue never runs dry: In every window, the jobs waiting in the queue request on average more GPUs than the whole fleet holds (360). The inference side therefore has spare nodes in the evening and at night, while Slurm still has work waiting for them.

Table 2: Workload by time of day on the weekdays of the replayed week (22 to 26 June 2026), per six-hour window. Inference: requests per hour answered by locally hosted models (requests forwarded to external APIs are left out), and the peak demand in nodes of the modeled 18-node inference side (Section 4.4). Slurm, for the A100 jobs the simulator replays: jobs submitted per hour, the GPUs held by running jobs, and the GPUs that jobs waiting in the queue request, the last two as means over the window. The fleet holds 360 GPUs, which caps the GPUs in use but not the queued demand: A backlog can ask for several times the hardware that exists. The queued column counts only jobs that later started, so the real backlog was at least this large. The weekend, a single day in this window, follows its own rhythm and is left out. Computed by `scripts/workload_by_window.py`.

	Kubernetes (inference)		Slurm (batch)		
Time window	requests per hour	peak demand (nodes)	jobs submitted per hour	GPUs in use	queued GPU demand
Night (00–06)	173	3.8	160	297	756
Morning (06–12)	2,262	17.8	369	286	594
Afternoon (12–18)	3,159	15.8	135	308	1,120
Evening (18–24)	855	6.0	206	307	1,011

Approach 1: The on-demand, Kubernetes-first rule - The first approach reacts to the observed state, with no schedule and no forecast. At every moment, the rule is:

1. **Default state.** All dynamic nodes sit in Kubernetes. The inference workload is the latency-sensitive one, and it should find a warm pool of nodes ready to serve.

2. **Slurm requests a node.** When a new Slurm job arrives that cannot start on any currently idle Slurm-side node, and the Kubernetes side has spare capacity, the service selects one dynamic node from the Kubernetes side and moves it to Slurm. If a dynamic Kubernetes node is idle at that moment, the move starts at once; if every dynamic node is serving requests, the service cordons one so that it takes no

new requests and moves it once its running requests have finished. No Kubernetes request is preempted.

3. **A lent node returns once it has been idle long enough.** Once a dynamic node has been idle in Slurm (no running job, no queued job that needs it) for at least the cooldown `IDLE_THRESHOLD`, the service returns it to Kubernetes. This rule consults no Kubernetes demand at all; its purpose is to keep nodes on the Kubernetes side whenever Slurm does not actively need them.

4. **Inference demand reclaims lent nodes (the demand check).** Every five minutes the service compares the real inference demand, plus a small reserve of spare capacity, with the capacity on the Kubernetes side. If the demand is larger, it takes lent nodes back: An idle one moves at once, and a busy one accepts no new job and moves once its running job ends (Section 5.3.1).

The approach is *Kubernetes-first* in the sense that Kubernetes is the home of every dynamic node unless Slurm has a concrete reason to borrow one. It is *on-demand*, or reactive, in the sense that no decision occurs until a Slurm job arrives, a node has been idle long enough, or the inference demand exceeds the remaining capacity.

The cooldown `IDLE_THRESHOLD` is the key parameter of this approach, because it prevents *thrashing*: the oscillation of a node between roles faster than either side can use it. The cooldown has to exceed the worst-case move time comfortably; otherwise a node that has just arrived in Slurm could return to Kubernetes before any Slurm job could start on it. The reference setup uses an `IDLE_THRESHOLD` of 300 s against a move time of roughly 120 s, a ratio of about 2.5×. The evaluation keeps this ratio fixed and scales the cooldown with the move time when it varies the latter (Section 5.7).

The rule has one weakness: It always acts one move late. A node starts moving only after a Slurm job is already waiting, and it returns to Kubernetes only after inference demand has already climbed. While a move lasts a couple of minutes this lag stays small, but the slower the move, the more it costs. The second approach addresses this lag.

Approach 2: The predictive policy - The second approach plans ahead. Instead of waiting for a side to run short, the service predicts the inference demand of the coming hour and sets the split before the change arrives.

At the start of each hour, for example at 12:00, the service takes the inference demand of the past seven days in clock-hour bins, the last one being 11:00 to 12:00, measured in GPUs as defined in Section 4.4. It asks a forecasting model for the demand of the coming hour, 12:00 to 13:00. The forecast uses only demand that has already occurred, so the plan never depends on data the service could not have had at that time. Within the hour, every 15 minutes (at 12:15, 12:30, and 12:45), the service makes no new forecast: It compares the current split with the plan from 12:00 and starts any move that is still missing, for example once a busy lent node has finished its job. The next forecast follows at 13:00. The forecaster is Chronos-Bolt (Section 3.8), a pretrained time series model applied as delivered, with no training on GWDG data [Ans+24]. It returns a range rather than a single guess, and the service deliberately reads the upper end of that range, the 90th percentile: Guessing too high costs a few lendable nodes, guessing too low costs inference capacity, so the service errs on the safe side. That upper band becomes a node target: the predicted GPUs, divided by four per node and rounded up, plus a safety floor of one spare node. Movable nodes above the target are lent to Slurm. Because the plan is set at the start of the hour and a node takes about two minutes to move, the nodes are

back on the inference side within the first minutes of the hour, instead of after the load has already risen.

As a safety net, the predictive policy also keeps rule 4 of the on-demand approach, the demand check: Every five minutes it checks the real inference demand and takes lent nodes back if the inference side runs short, for example when the demand rises faster than the forecast expected.

Figure 13 shows the two approaches side by side on three real days from the inference trace. The forecast band follows the real demand closely (over the simulated week, the true demand fell inside the band in 74% of the hours and below its upper edge in 85%). The predictive node target steps down in the evening and climbs back before the morning rise. The on-demand target, computed here from the previous hour's demand, makes the same moves an hour later, after the demand has already changed. The nodes between a policy's step line and the 18 nodes of the full inference side are the ones it lends to Slurm during those hours.

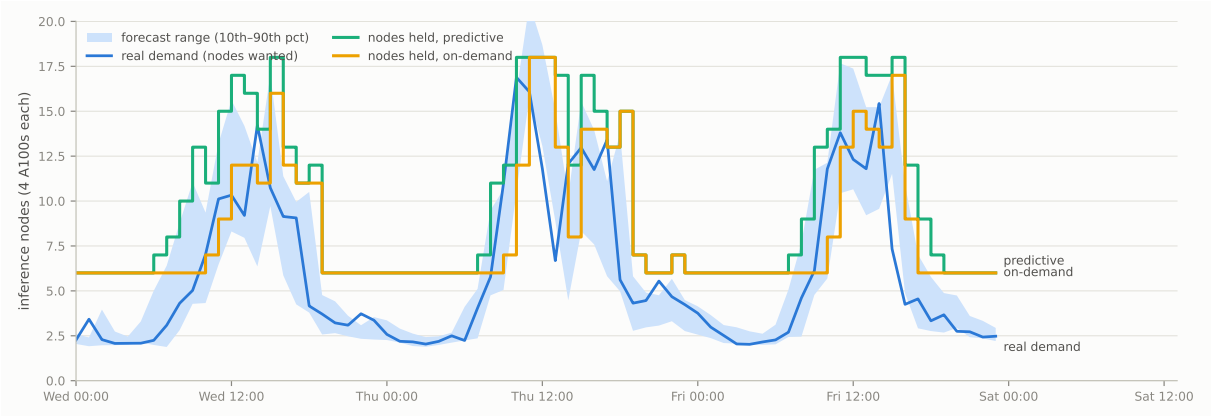


Figure 13: The two policies on three real days from the inference trace (24–26 June 2026). The blue line is the real demand, in nodes; the shaded band is the rolling Chronos-Bolt forecast range (10th to 90th percentile), each hour predicted only from the hours before it. The green steps are the nodes the predictive policy holds (90th percentile plus one spare node); the yellow steps are what the on-demand rule holds, which lags the demand by about an hour on every rise and fall.

Fixed schedule as the simplest forecast - One special case is worth naming, because it needs no model at all. If the daily pattern is stable, the forecast can be written by hand once, as a fixed schedule derived from past traces, for example: Lend the dynamic pool to Slurm late in the evening and reclaim it before the working day begins. At each schedule boundary the service performs the five-phase move from Section 4.2.4 on every node whose side changes, starting a few minutes early, so that the reboots finish before the window begins. Such a schedule is predictable and easy to audit, but it is a forecast frozen at write time: It cannot notice a public holiday, a conference deadline, or any other day that breaks the pattern. The full predictive policy re-plans every cycle from fresh data, so it adapts where the schedule cannot.

Fixed nodes in both approaches - Whichever approach is in force, the service never touches the fixed Kubernetes and fixed Slurm nodes. The Kubernetes control plane runs on its fixed control-plane nodes; the Slurm controller and one fixed compute node remain on the Slurm side at all times. These fixed nodes guarantee each cluster a

minimum capacity even when the dynamic pool sits entirely on the other side. An operator can raise these minimums by tagging more nodes as `cluster=k8s` or `cluster=slurm` in `node-config.conf`.

Choosing between the two approaches - The on-demand rule requires no history and no model, so it is the appropriate starting point for a site without a stable load pattern or without enough trace data to forecast from. The predictive policy pays off where the load follows a clear daily cycle, as the GWDG traces show it does, and it pays off more the longer a move lasts, because it avoids reacting late. Section 5 measures both approaches, and the static baseline, against the same real traces, so that an operator can compare them in numbers.

4.2.6 Design rationale

This section states the reasons behind the design choices that were not self-evident.

Reboot, not in-place reconfiguration - An alternative design would retain both `slurmd` and the kubelet on every dynamic node and switch roles by starting and stopping daemons, without a reboot. The thesis rejects this path for three reasons. First, leftover state would survive across roles in places that are hard to audit: iptables rules from `kube-proxy`, mounted ConfigMap secrets, container runtime images, and cgroup hierarchies the two systems arrange differently. Decker and Kunkel give the same reason for Ephemeral Kubernetes itself [DK25]. Second, in-place switching forces both stacks into the same Warewulf image; the image grows, and a Slurm-only update has to wait for a Kubernetes rebuild. Third, reboot-based switching is the same operation Warewulf already performs for routine software upgrades, so the negotiation service reuses a path the site already operates. The price is the boot time itself, which Section 4.2.4 treats as a central constraint.

A single daemon instead of per-node agents - A distributed design would place a small agent on every dynamic node and coordinate through a quorum protocol or a shared key-value store. The thesis chooses a single daemon on the Warewulf host, because the rights that daemon needs (reading state from both schedulers, writing state to the provisioning system) are already concentrated there: The host owns the images, the overlays, the DHCP and TFTP configuration, and the credential server. A distributed design would either replicate that authority onto every node, which raises the attack surface Section 3.5.6 warned about, or shift the same decisions onto yet another control-plane component for no benefit. If the daemon itself fails, both clusters continue on the nodes they currently own; the dynamic pool freezes in its present split until the daemon restarts, so a daemon failure costs flexibility but no running work.

Every dynamic node pre-listed in `slurm.conf` - The alternative would be Slurm's *cloud nodes* feature (Section 2.3.9) or a `scontrol create node` call at entry. Both work, but both require Slurm to learn the node's hardware at runtime, so a Slurm-side configuration error would block a move that would otherwise succeed. Pre-listing replaces this with a one-time `slurm.conf` generation step (in `03-provision-slurm.sh`); afterwards the controller already knows every node that might ever join.

Thrashing suppression through the cooldown - A reactive policy can in principle oscillate a node between roles faster than either side can use it. The design relies on one mechanism against this: the `IDLE_THRESHOLD` cooldown from Section 4.2.5. A node that has just arrived in Slurm cannot return until it has been idle in Slurm for at least `IDLE_`

`THRESHOLD` seconds, and a node moved to Kubernetes does not return to Slurm until a newly arrived Slurm job cannot start anywhere else. Together these two rules set a lower bound on the time a node needs for a round trip.

Section 4.2.7 presents the reference implementation of this design, and Section 5 measures the two approaches of Section 4.2.5 with the metrics of Section 4.3.

4.2.7 Implementation

The reference implementation is deliberately small: about 1,600 lines of Bash in four scripts plus a shared library, one short Python helper, and one systemd unit, all installed on the Warewulf host and nowhere else. Table 3 lists the parts and the task of each.

Table 3: The files of the reference implementation and what each one does.

File	Lines	Task
`common.sh`	166	Shared library: reads the two configuration files, filters nodes by cluster and role, rewrites the `current_cluster` field, and provides the logging functions. Every script sources it, so that all parts read the same state in the same way.
`01-base-install.sh`	417	One-time setup: installs Warewulf, builds the two images and the two overlays, and starts the credential file server.
`02-provision-k8s.sh`	101	Registers one Kubernetes node in Warewulf, generates its join key, and boots it.
`03-provision-slurm.sh`	222	Registers one Slurm node, generates `slurm.conf` with every dynamic node pre-listed, and boots it.
`04-balancer.sh`	702	The negotiation daemon: the five-phase move of Section 4.2.4, the decision loop, and the timing records.
`ww-file-server.py`	91	The credential file server from the security model [Mad26], which hands the cluster credentials to a booting node that presents its per-node key.
`hybrid-balancer.service`	18	The systemd unit that starts the daemon.

Choice of implementation language - The service is glue code: Every step it performs is a command that already exists (`wwctl`, `kubectl`, `scontrol`, `squeue`, `ssh`, `reboot`). Implementing the daemon in Bash keeps each step literally the command an administrator would type by hand, so the log reads as a transcript, any step can be replayed manually when something goes wrong, and the Warewulf host requires no additional language runtime or libraries. The one task Bash suits badly, serving credentials over HTTPS, runs in the short Python helper instead. The forecasting model of the predictive policy is also Python, because Chronos-Bolt ships as a Python library [Ans+24]; the daemon consumes only its output, a plain node count.

Division of responsibilities - The daemon implements the machinery: draining either side, reassigning image and overlay, rebooting, verifying the join, and updating the state file. The moves are separated from the decisions: One function receives a node and a target side and performs the move, so a policy only has to choose which node goes to which side. The checked-in daemon drives this function on the simplest possible schedule, alternating each dynamic node between the two roles on a fixed cycle (five minutes in the

reference setup); this mode exists to exercise the move machinery repeatedly and to collect the transition measurements of Section 4.2.4. The two decision approaches of Section 4.2.5 attach to the same function, and the simulator of Section 4.4 measures their behavior on real demand against the GWDG traces.

Setup and restart behavior - At every start the daemon first checks which nodes from `node-config.conf` are already registered in Warewulf and provisions the missing ones through the two provisioning scripts, control-plane and controller nodes first. The check is idempotent: On a machine where everything is already in place it performs no action, so the same command sets up a hybrid cluster on a fresh Warewulf host and restarts cleanly on one that is already running. Under systemd, the unit starts after `warewulfd`, restarts on failure after a 30-second pause, and writes to the journal. Because the only state that matters lives in `node-config.conf` (Figure 9) and is updated as part of every move, a restarted daemon re-reads the file and carries on from the recorded state.

Two configuration files - `node-config.conf`, described in Section 4.2.2, lists the nodes; a second file, `cluster-settings.conf`, describes the site: the manager's address and DHCP range, the Slurm version, and the hardware line for `slurm.conf` (CPUs, sockets, memory), measured once with `slurmd -C` on a real compute node and inserted into every generated configuration. Both files are plain text with `#` comments, and an operator edits them directly; the `current_cluster` field is the only entry the service itself ever rewrites.

Logging and measurement - The daemon writes three records. First, a plain log under `/var/log/hybrid-cluster/`: Every external command is recorded before execution and its output appended, so that the log is a complete transcript of what the service did. Second, the per-move timing files from Section 4.2.4: `boot-times.csv` holds one machine-readable row per move (the clock times of the reboot command, the first successful ping, and the verified cluster join, the durations between them, and a success or timeout flag), and a companion human-readable log holds the same facts as text. This CSV is the raw data behind the transition-time results of Section 5.1. Third, the systemd journal, for the usual `journalctl` view. The join checks poll the new side under fixed limits (five minutes for a reboot and a Kubernetes join, three for a Slurm join); a move that exceeds its limit is recorded as a timeout and, as Section 4.2.3 explained, left to the operator rather than retried blindly.

Test modes - The daemon provides two switches for safe testing. With `--dry-run` it prints every command it would issue and executes none of them, which exposes the full decision and move sequence without touching a node. With `--once` it performs exactly one cycle and exits. We developed and debugged the move sequence in these modes before it ever rebooted a real node, and both remain useful when the configuration changes.

4.3 Metrics

Table 4 defines the six metrics the evaluation collects. Slurm queue wait, Kubernetes service level, idle capacity, and forecast accuracy answer RQ3; transition time and transition count answer RQ4; whether the prototype completes a full end-to-end move in both directions answers RQ1 and RQ2.

How each metric is computed - The Slurm queue wait is the interval between the submission of a job and its start, reported as mean, median, and 95th percentile over all replayed jobs. The Kubernetes service level is measured on the requests themselves:

Table 4: Metrics collected in the evaluation and the research questions they address.

Metric	Quantity measured	Addresses
Transition time	Total time from the drain decision to the node appearing `Ready` (Kubernetes) or `IDLE` (Slurm) on the new side. Reported as median and 95th percentile per direction.	RQ1, RQ2, RQ4
Kubernetes service level	The inference requests that have to wait for capacity instead of starting at once: how many, the mean wait over all requests, and the longest wait. Reported per policy.	RQ3
Slurm job queue wait time	Time from a job's arrival to its start. Reported as median, mean, and p95 per policy.	RQ3
Idle capacity	The node-hours in which capacity serves no job or request: Slurm nodes that run no job, inference capacity above the demand, and nodes in transition. Reported per side and for the whole fleet, per policy.	RQ3
Transition count	The number of transitions from Kubernetes to Slurm and back over the simulation window. A high count relative to the workload signals oscillation.	RQ4
Forecast accuracy	The distance between the predicted inference demand and the real one, reported as MAE, sMAPE, and MASE against simple seasonal baselines.	RQ3

Every request of the trace is replayed at the time it really arrived and starts at once if the inference side has room; if it does not, the request waits, and the simulator records that wait. The service rejects no request, so the inference columns of Section 5 state how many requests were delayed, the mean wait over all requests (including those that started at once), and the longest wait. Idle capacity is counted from one sample per minute: A Slurm node is idle while it runs no job; the inference side counts the capacity above the demand of that minute as idle, because the simulator treats that side as one pool; and a node in transition is idle, because it serves neither side. The transition count is the number of role changes over the replay window, since each one costs a reboot and a high count indicates an unstable policy. Forecast accuracy is reported twice: as the share of hours in which the true demand stayed below the 90th-percentile forecast, and as the error of the median forecast relative to the simple baselines (MASE).

4.4 Experiments

The evaluation employs a discrete-event simulator that replays the two workloads on one shared GPU pool. It compares five configurations of that pool under the same workloads: a static baseline that never moves a node (Figure 12), the on-demand policy, the predictive policy (both in Section 4.2.5), the fixed schedule (the hand-written special case of Section 4.2.5), and an oracle that reads the future demand directly from the trace. The oracle cannot be built in practice; it marks the best result any policy could reach. Applying all five to the same input separates the contribution of the node sharing itself

from what the prediction adds on top.

Slurm workload - For the Slurm side the simulator generates no jobs; it replays a real trace of past jobs. The trace is a six-day `sacct` export from the GWDG Grete cluster covering 22–27 June 2026, with 84,923 jobs from 481 users. The evaluation retains every NVIDIA A100 GPU job, 26,667 in all, and replays each one at its real submit time, holding its GPUs for the job's real duration. Most jobs request a single GPU (91% of the GPU jobs), so several share a node; the simulator packs them onto four-GPU nodes, as the shared GPU partition of Grete does, and a node returns to the movable pool only once all four of its GPUs are free. The arrival bursts, the mix of many short jobs and a few long ones (the median job lasts about four minutes, the mean over forty), and the queue pressure they create all originate in real data.

Inference workload - Real data drives the Kubernetes side as well. A month-long export from the SAIA inference platform (1 to 29 June 2026, Section 5.2) holds one row per request with the model that answered it and its start and end time, about 936 thousand requests in all. Just under a third of them are forwarded to external APIs and never touch a local GPU, so the evaluation discards them. From the rest it builds a demand curve $D(t)$: the number of A100 GPUs the inference service needs at time t. Each request in flight at time t contributes the GPU share of its model, taken from a footprint table that maps every served model to the GPUs one serving instance occupies and to the number of requests such an instance carries at once. That table has to be derived, because the export records the requests but not how the cluster served them: It states neither how many instances of each model were active, nor on how many nodes, nor with how much GPU memory. Section 5.2 explains the deduction and Table 9 lists the result per model. Counting GPUs instead of raw requests weights each request by its cost: A request against a 675-billion-parameter model spread over eight GPUs weighs far more than a burst of sub-second pings to a small 8-billion-parameter one. Reserving enough whole instances for every model's observed peak takes 62 GPUs, or 16 four-GPU nodes. The demand curve stays far below that, because it charges each request only its share of an instance, and an instance is rarely full: A model with a single request in flight still reserves all of its GPUs. That difference between reserved and used capacity is what the negotiation service can lend out. Because the export covers the days of the Slurm trace, the simulator feeds both sides the real demand of the same calendar days, lined up day for day. The evaluated days fall in the last week of the month-long inference trace, so the forecaster has about three weeks of earlier data for its rolling seven-day history.

Forecasting protocol - The predictive policy requires a forecast of $D(t)$, and the evaluation tests that forecast before relying on it. The test is rolling: The trace is divided into three-hour bins, and from the eighth day on, each bin is predicted from the seven days before it only, so the model never sees the value it predicts. Four simple baselines set the reference: the last observed bin, the same bin yesterday, the same bin a week ago, and the average of the same bin over the last seven days [HA21]. Chronos-Bolt [Ans+24], applied zero-shot, competes against them. Accuracy is reported as MAE, sMAPE, and MASE, where MASE is scaled so that "same bin yesterday" equals 1.0 and any lower value is more accurate [HK06]. The Slurm side needs no forecast, as its trace shows: Its GPU demand is nearly flat across the day (the daytime average lies only about 4% above the overnight average), and its queue is backlogged around the clock, so any node handed to Slurm is occupied almost at once.

Cluster setup - The simulated hardware is a shared pool of 90 A100 nodes, the size of the Grete GPU fleet on which the trace was recorded, with four A100 GPUs on each node (no single-node job in the trace requests more than four). A node is the movable unit: It is re-imaged as a whole, so it can cross between Slurm and Kubernetes only when all four of its GPUs are free. The pool is divided into 72 fixed Slurm nodes, 6 fixed Kubernetes nodes, and 12 movable nodes, so the movable pool is about an eighth of the fleet (Table 5). We chose 72 fixed Slurm nodes because with them the no-sharing baseline reproduces GWDG's real queue, a median wait of about two to three hours. The trace itself shows a median wait of about two hours, and it understates the real pressure, because it holds only the jobs that eventually started. The inference side is sized from its demand curve, which the simulator scales so that its busiest moment fills 18 nodes (Section 6.1 explains this scaling): The 6 fixed nodes cover the overnight low, and the 12 movable nodes cover the daytime peak. The move time is 2 minutes and the cooldown `IDLE_THRESHOLD` 5 minutes. One further rule protects both sides: A lent node accepts only jobs of up to four hours, which real Slurm enforces as a partition time limit, so that no reclaim can wait longer than that.

Table 5: Fleet and policy parameters of the trace-driven simulation. The fleet mirrors the A100 partition on which the Slurm trace was recorded. The policy parameters are the defaults of Section 5.3; Section 5.7 varies four of them. The transition time of 2 minutes is the assumption derived from the prototype measurements of Section 5.1.

Simulated fleet	
Pool	90 nodes, 4 A100-80 GB GPUs each (360 GPUs)
Split	72 fixed Slurm, 6 fixed Kubernetes, 12 movable
Replay window	Monday 22 to Saturday 27 June 2026 (6 days)
Workload in the window	26,667 A100 Slurm jobs, 203,691 inference requests
Policy parameters	
Transition time	120 s per node, plus the wait for running work to finish
Cooldown	300 s idle in Slurm before a lent node returns
Time limit on lent nodes	4 h, enforced as a Slurm partition limit
Forecast	Chronos-Bolt (base), 1-hour bins, 7-day rolling context, one bin ahead
Provisioning quantile	p90, plus a safety floor of 1 node
Check intervals	plan check every 900 s (the forecast itself is made once per hour); demand check every 300 s

Prototype testbed - The 2-minute transition time in the simulator is based on measurements from a prototype, which also answer the first part of RQ4, the cost of one transition (Section 5.1). We measured the transitions on a virtual cluster hosted on GWDG's OpenStack cloud, listed with its version numbers in Table 6. The Warewulf manager, the Slurm controller, the Kubernetes control plane, and the compute nodes were separate virtual machines on one tenant network, so every node booted over the network from Warewulf just as a physical node does. The virtual machines carry no GPU. This does not change which steps a transition runs, because the image Warewulf assigns decides the node's role and neither the drain nor the join step inspects an accelerator. It does make the boot phase shorter than on a production GPU node, a limitation Section 5.1 returns to. The daemon executed in the fixed-cycle test mode of Section 4.2.7, which alternates a dynamic node between the two roles, and it recorded the duration of every

transition, twenty in all, ten in each direction. No node carried work when its transition began, so the times include no wait for a running job or pod to finish.

Table 6: Hardware and software of the prototype testbed on which the transition times of Table 8 were measured. Every machine is a virtual machine on GWDG's OpenStack cloud; the compute-node line is the hardware description that `slurmd -C` reported and that the generated `slurm.conf` carries. The testbed holds no GPU, so it measures the provisioning path rather than the boot time of a production accelerator node (Section 6.1).

Component	Configuration
Platform	GWDG OpenStack cloud, KVM virtual machines on one tenant network
Machines	1 Warewulf manager, 1 Slurm controller, 1 Kubernetes control plane, dynamic compute nodes
Compute node	2 vCPU (2 sockets, 1 core per socket), 7.9 GB RAM, no GPU, no persistent local disk
Node image	Rocky Linux 9, built from `ghcr.io/warewulf/warewulf-rockylinux:9`, held in RAM
Provisioning	Warewulf 4.5.8
Batch scheduler	Slurm 24.05.4, built from the SchedMD source RPMs
Kubernetes	v1.32.1 (`kubeadm`, `kubelet`, `kube-proxy`), etcd 3.5.24-0
Container runtime	`containerd` with the `systemd` cgroup driver
CNI plugin	Flannel v0.26.4
Negotiation service	`hybrid-balancer.service` on the Warewulf manager, Bash

Software and code revisions - Table 7 names every component behind the results, with the version or the commit used, each commit linked directly to that revision in its repository. Two code bases carry the prototype. The first is the negotiation service, the reference implementation of Section 4.2.7.[8] The second is Ephemeral Kubernetes [DK25], of which the deployment uses a GWDG-maintained fork that tracks current Kubernetes releases, on branch `main`.[9] The simulation, the forecast, and the figures come from the analysis scripts of the thesis repository.[10] Pinning the commit of each repository, rather than only its address, allows a reader to reconstruct the exact state that produced the numbers reported here.

Sensitivity tests - The evaluation varies four parameters (Section 5.7). First, the size of the dynamic pool, from zero, which reproduces the static baseline, up to the whole inference side, which shows the trade-off between Slurm queue wait and inference service level. Second, the provisioning quantile of the predictive policy, from the 50th to the 90th percentile, the highest band the forecaster provides: A low quantile lends Slurm more nodes but risks leaving the inference side short, and a high one is safe but lends less. Third, the safety floor, from zero to four spare nodes on top of the forecast. Fourth, the move time, from 2 to 60 minutes, with the cooldown scaled to match: Where moves are fast,

[8]Negotiation service: `https://gitlab.gwdg.de/shrinath.madde/dynamically-negotiating-resources-between-kubernetes-and-slurm-environments`, accessed 16 April 2026.

[9]Ephemeral Kubernetes fork: `https://github.com/gwdg/pub-2025-ephemeral-kubernetes`, accessed 2 April 2026.

[10]Thesis repository, including the analysis and simulation scripts: `https://github.com/shrinathmadde/Masters-Thesis`, accessed 20 July 2026.

Table 7: Software, code repositories, and the exact revisions behind the results of this chapter. The last column gives the abbreviated Git commit deployed on the testbed of Table 6 or used to produce the simulated results; the three repositories are cited in the footnotes of the preceding paragraph. The Python packages are those of the analysis environment, and Table 6 lists the cluster software of the prototype separately.

Component	Source	Version or commit
Code written for this thesis		
Negotiation service	GWDG GitLab, branch `main`	`2cc27b5`
Ephemeral Kubernetes	GWDG GitHub fork of the reference implementation [DK25], branch `main`	`34feec7`
Simulation and analysis	Thesis repository, directory `scripts/`	`fa4bb79`
Analysis environment		
Python	CPython	3.12.3
NumPy	`numpy`	2.5.1
pandas	`pandas`	3.0.3
Matplotlib	`matplotlib`	3.11.0
Forecasting library	`chronos-forecasting`	2.3.1
Forecasting model	`amazon/chronos-bolt-base`	as published
Tensor runtime	PyTorch (CPU build)	2.12.1

reacting on demand is nearly as good as predicting; the slower the move, the more the forecast is worth.

Simulator assumptions - The simulator makes three simplifications. First, the simulator schedules Slurm jobs first-come-first-served, without backfill, because the trace does not record the time limits backfill requires; this makes the queue waits an upper bound. Second, the inference side is modeled as fungible capacity: A request needs GPU share somewhere on the Kubernetes side, not on a specific node. Third, the move time is one fixed value per run and covers only the switch itself: reassigning the image, rebooting, and joining the new side. The wait for a running job to finish before a busy node can move is not part of this value; the simulator takes it from the real job runtimes in the trace. The measured transition times for RQ4 come from the prototype, not from the simulator.

Output - The simulator records a queue wait for every Slurm job, a wait for every inference request (zero if it started at once), the capacity provided and required on the Kubernetes side at fixed time steps, the idle node count on each side, and a time-stamped log of every move with its direction. These records are the raw data analyzed in Section 5.

5 Evaluation

This chapter reports the evidence for the four research questions of Section 4.1: Whether one Warewulf-managed cluster can provision the same node in either role (RQ1), whether the service switches that role unattended and without disrupting running work (RQ2), whether automatic sharing shortens the Slurm queue without degrading the inference service (RQ3), and what a single transition costs (RQ4). A mixed workload of this size could not be placed on GWDG's production systems, so we measured the transitions on a prototype testbed and replayed two production traces through the discrete-event simulator, both described in Section 4.4 together with the simulated fleet, the policy parameters, and the software revisions behind the results that follow. Section 5.1 reports the measured cost of one transition, the one quantity the simulator cannot derive and the first part of RQ4. Section 5.2 characterizes the two traces and shows that inference demand and batch demand peak at different times of day, which is the gap the negotiation service exploits. Section 5.3 defines, for each policy, the rule that triggers a transition, together with the safety rules that protect the inference side; the metrics themselves are defined in Section 4.3. Section 5.4 quantifies how accurately Chronos-Bolt predicts the inference load. Section 5.5 presents the main result for RQ3: How far each policy shortens the Slurm queue wait, how close the predictive policy comes to an oracle forecast, what each policy costs the inference service, and how much idle capacity each one removes. Section 5.6 follows a single day to show when the transitions occur and how many nodes move at once, which answers the second part of RQ4, the stability of the system. Section 5.7 varies the size of the movable pool, the provisioning quantile, the safety floor, and the transition time. Section 5.8 closes the chapter by answering the four research questions in turn; Section 6 opens by stating what these numbers do and do not generalize to.

5.1 Transition cost

Every other number in this chapter comes from the simulator, but the simulator cannot derive one quantity: the time a node needs to change sides. The prototype of Section 4.2.7 supplies it, measured on the testbed described in Section 4.4, and the same measurement answers the first part of RQ4. A transition runs through the five steps of Section 4.2.4: The service drains the node on its current side, reassigns its image and overlay in Warewulf, and reboots it; the node then boots its new image over the network and joins the other cluster, and the transition ends once the service has confirmed the join. Table 8 gives the total time over twenty transitions, ten in each direction.

Table 8: Measured time for one node to change sides, from the start of the drain to the confirmed join on the new side, over twenty transitions of the prototype (ten in each direction) on the testbed of Table 6. The nodes carried no work when a transition began, so the times include no wait for a running job or pod to finish.

Direction	Median (s)	95th percentile (s)
Kubernetes to Slurm	98	121
Slurm to Kubernetes	102	128

A transition took a median of 98 seconds from Kubernetes to Slurm and 102 seconds back. The 95th percentile lay about 25 seconds higher in both directions, so no transition

took much longer than the others. Both directions cost about the same, because most of the time goes into the reboot, which does the same work either way: the firmware restart, the network boot from Warewulf, and loading the new image into RAM. The result is close to the under 90 seconds Decker and Kunkel report for assembling a five-node Kubernetes cluster from cold [DK25].

Two limits apply to these numbers. First, the nodes were idle, so the times do not include the wait for a running batch job or inference pod to finish; the simulation takes that wait from the real job runtimes in the trace instead (Section 4.4). Second, the testbed nodes are virtual machines without a GPU, so the times are a lower bound: A production GPU node initializes more firmware and loads a larger image, and we expect 5 to 10 minutes per transition on hardware such as GWDG's Grete nodes, an estimate this thesis did not measure.

The simulation assumes 2 minutes per transition, the measured medians rounded up to the next full minute. Production nodes are probably slower, so Section 5.7 repeats the simulation with slower transitions, from 5 up to 60 minutes each. Even at 5 or 15 minutes per transition the results barely change: The mean Slurm wait of every policy differs by at most 13 minutes from the result with 2 minutes, and the predictive policy still lets more than 99% of inference requests start without waiting. Slower transitions hurt the on-demand rule, which waits for the demand, more than the predictive policy, which starts its moves before the demand arrives, but they do not change the main result of this chapter.

5.2 Workload characterization

The evaluation draws on two traces from GWDG's production systems. The first is a month-long export from the SAIA inference platform [Doo+25], which serves LLM requests to GWDG's Chat AI users (Section 3.6): 935,774 requests from 22,617 users across 54 model endpoints, 1 to 29 June 2026, each row carrying the model that answered the request and its start and end time. The second is the six-day `sacct` export from the Grete GPU cluster of Section 4.4, whose fully covered window spans 22 to 27 June 2026 (84,923 jobs from 481 users). The two windows overlap, so the replay aligns the two workloads day for day rather than by their average shape.

Inference demand follows a strong daily cycle. Figure 14 plots the arrivals in hourly bins across the month. Traffic climbs through the morning, peaks around midday, and falls close to zero overnight, and the weekends stay quieter throughout. Grouped by hour of day (Figure 15), the busiest hour near 12:00 carries almost sixty times the load of the quietest hour near 03:00, the working day from 08:00 to 20:00 is about five times busier than the night, and a weekday is well over twice as busy as a weekend day. The requests themselves are short, with a median of eight seconds and a mean of thirty. Unlike a batch job, a request has to be answered within seconds, so the inference side must hold enough capacity for all the requests that run at the same time.

The export names the model that answered each request, but it does not record how the cluster served those requests. It holds no instance counts, no GPU counts, and no memory figures, and the platform could not be queried for them. We therefore derived them, and Table 9 gives the result.

Choice of GPU model - Every row of the table is sized on the A100 with 80 GB, four to a node. That is the card the Slurm trace consists of: 52,729 of its 54,615 GPU job records request `gpu:a100`. Because the negotiation service moves the same node between

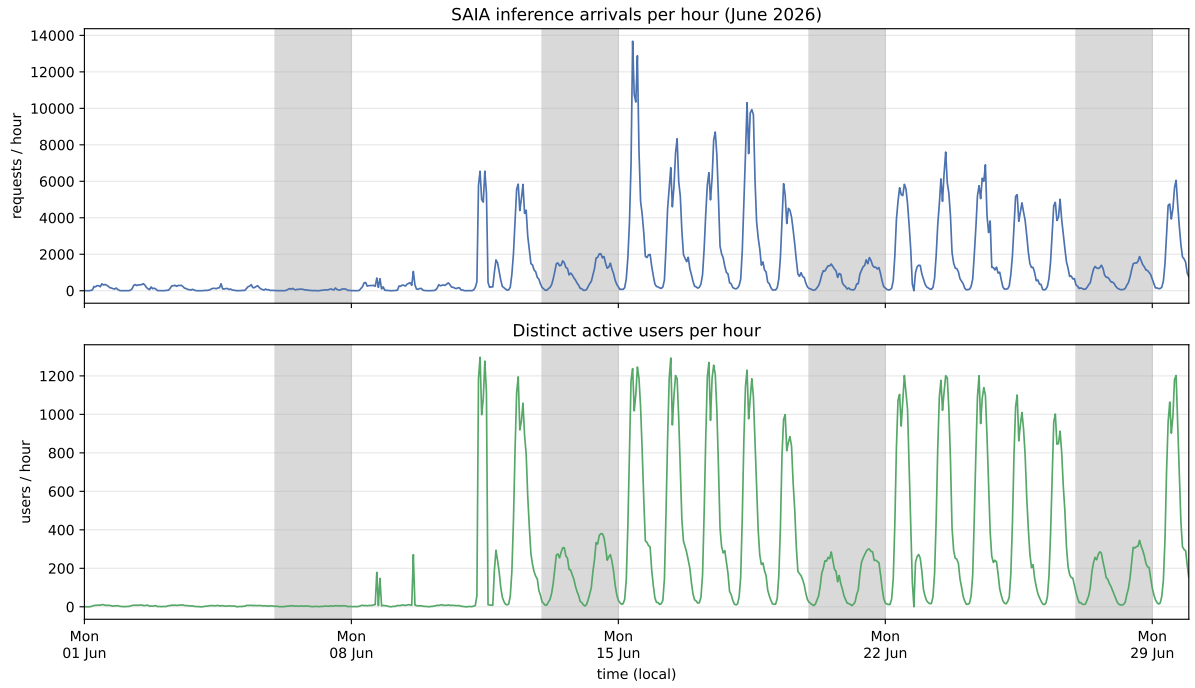


Figure 14: Inference requests per hour over June 2026, with the distinct active users per hour below. The gray bands mark the weekends. Traffic follows a sharp daily cycle and drops at the weekends.

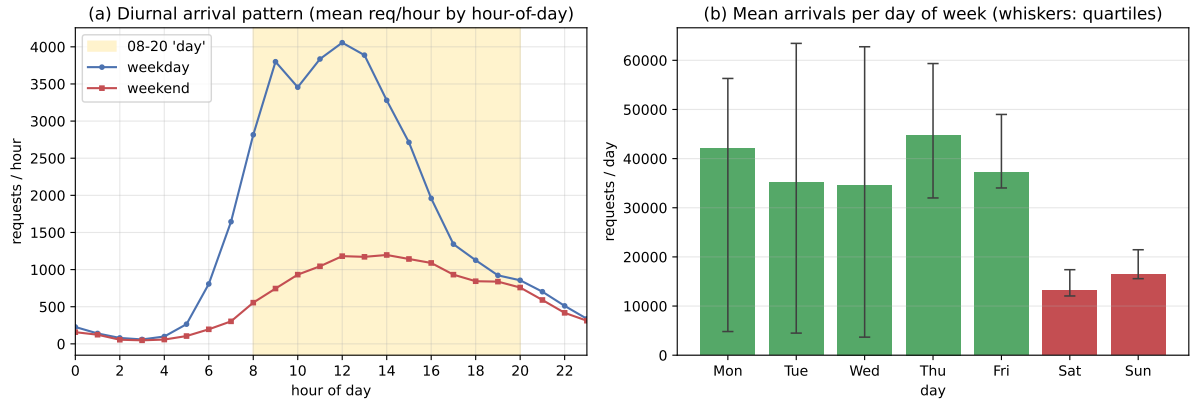


Figure 15: (a) Mean inference requests per hour of day, weekday against weekend; the shaded band marks the daytime window from 08:00 to 20:00. (b) Mean requests per day of week; the whiskers give the lower and upper quartile of the daily totals across the month.

both sides, the inference side has to be priced on the same hardware as the batch side; otherwise a node would change size on its way across. The table therefore states what the observed load would cost on the modeled A100 fleet, not what the live platform spends on its own hardware.

Derivation of each row - Three steps. First, weight memory: the parameter count times the bytes per parameter of the precision we assume for the model, 16, 8, or 4 bits. Second, cards per instance: the smallest group of cards whose memory holds the weights and still leaves a quarter of the group's memory free for the KV cache and the runtime.

The group sizes are powers of two (1, 2, 4, 8, 16), as is usual for tensor parallelism, which splits the attention heads evenly across the cards. Third, instances: the peak concurrency the trace shows for that model divided by the requests one instance carries at once, rounded up, and at least one.

Mistral Large 3 illustrates the derivation. Its 675 billion parameters at 4 bits weigh 338 GB. A group of four A100s holds 320 GB, which is not enough, so the group grows to eight cards, 640 GB, leaving 302 GB for the KV cache. The trace never shows more than 32 of its requests in flight together and one instance carries 64, so a single instance suffices and eight GPUs stay reserved. Dividing those eight GPUs by the 64 requests means that each request in flight counts as an eighth of a card. That per-request share is the weight the demand curve of Section 4.4 charges for every request in flight, for as long as it lasts, and it ranges from under a hundredth of a card for the default chat model to an eighth for the largest.

Serving the observed peak requires 62 GPUs, or 16 four-GPU nodes, and only the busiest model, the default chat model, needs more than one serving instance. The average columns show the same pattern as the daily cycle: Short peaks set the reserved capacity, and most of it sits idle most of the time. The least certain column is the number of requests one instance carries. vLLM accepts up to 256 requests per instance by default, but the KV-cache memory runs out well before that, so we assume 128 for models with small active parameter counts and 64 for large ones. That is an estimate, not a measurement, and so are the precisions. Neither has to match the live deployment, because the negotiation service never sees a model: It sees one demand curve, and the simulator multiplies that whole curve by a single constant to size the service against the modeled hardware (Section 6.1). If every model's weight were off by the same factor, for example because all of them are served at twice the precision, that constant would absorb the error; only a change in the models' weights relative to each other changes the shape of the curve. The table as a whole therefore estimates the least capacity the observed load requires, not a measured deployment.

Slurm GPU demand looks very different. Figure 16 plots the number of GPUs held by running batch jobs, averaged over each hour of the day across the covered window (Monday 22 to Saturday 27 June). The line is almost flat: The daytime average lies only four percent above the overnight average, and the busiest hour of the day reaches less than 1.2 times the quietest. The queue never empties, so a GPU freed by one job is soon taken by the next, and the pool stays occupied day and night. The jobs hold 314 GPUs on average, about 79 four-GPU nodes, and occupancy of the A100 fleet peaks at 92 nodes, close to the 90-node fleet the simulator models. The one weekend day in the window, the Saturday, is about a third quieter than the weekdays.

The two shapes complement each other. When inference traffic falls overnight and at the weekend, the GPUs reserved for it sit idle while Slurm still holds a full queue of batch work. A service that lends those idle inference nodes to Slurm during the quiet hours and reclaims them before the next working day would recover GPU time that is otherwise wasted. The rest of the chapter quantifies what that is worth. Because the inference export spans a whole month, the replay draws on the real inference demand of the very days the Slurm trace covers (22 to 27 June) rather than on an averaged shape from another week. That week also stresses the policies: Its weekday mornings ramp steeply, from two or three nodes to the full eighteen within a few hours, so the timing of the reclaim matters.

Table 9: Estimated GPU footprint of the self-hosted inference services on the modeled A100 fleet, derived from parameter counts, an assumed serving precision, and the request concurrency measured in the month-long trace. GPUs are 80-GB A100 cards, four to a node, matching the hardware of the Slurm trace, so the table describes the modeled fleet rather than the live deployment. Cards per instance is the smallest power-of-two tensor-parallel group that holds the weights and leaves a quarter of its memory for the KV cache. "One instance" gives those cards and the concurrent requests one instance carries; their ratio, GPU per request, is the weight one request in flight carries in the simulation's demand curve. The total of 62 GPUs (16 four-GPU nodes) is the least capacity that serves the observed peak; the trace records no instance or GPU counts. Two services with fewer than ten requests in the month are omitted. *The default chat model, the only one that needs more than one instance.

	Concurrent requests			One instance			
Model	peak	average	Weight memory (GB)	GPUs	requests	GPU per request	GPUs reserved
Qwen 3 30B A3B*	258	2.7	30	1	128	0.008	3
GPT-OSS 120B	55	0.5	64	2	64	0.031	2
Qwen 3.6 35B A3B	48	0.4	35	1	128	0.008	1
Qwen 3.5 122B A10B	44	0.5	122	4	64	0.063	4
Qwen 3.5 397B A17B	42	1.2	397	8	64	0.125	8
Mistral Large 3 675B	32	0.9	338	8	64	0.125	8
GLM 4.7	32	0.3	355	8	64	0.125	8
Devstral 2 123B	22	0.1	123	4	64	0.063	4
Apertus 70B	21	0.1	140	4	64	0.063	4
DeepSeek R1 Distill 70B	16	0.2	140	4	64	0.063	4
Gemma 4 31B	13	0.3	62	2	64	0.031	2
13 smaller services	≤34	0.6	0–54	1–2	8–128	0.008–0.125	14
Total							62

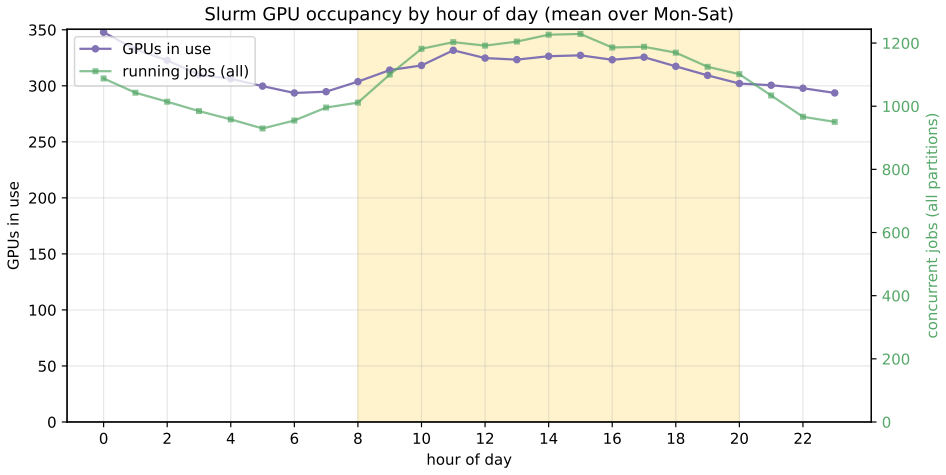


Figure 16: A100 GPUs held by running Slurm jobs, averaged over each hour of the day (Monday to Saturday). Unlike inference, batch demand stays almost flat around the clock.

5.3 Policy definitions

The simulator replays the two traces as a discrete-event model. Slurm jobs arrive at their real submit times and hold their GPUs for their real run times, and the inference demand rises and falls with the real request load. All five policies operate on this same replay, and every policy that moves nodes carries them through the same five phases of Section 4.2.4. They differ in one respect only: When they decide to start a move. This section states that decision for each policy, so that the numbers later in the chapter can be read against the exact rules that produced them.

Throughout, the 90 nodes fall into three groups. The 72 fixed Slurm nodes and the 6 fixed Kubernetes nodes never move; they keep each cluster working even when the whole movable pool is on the other side. Only the 12 dynamic nodes move, and a dynamic node can move only when all four of its GPUs are free, because a move re-images the whole machine.

No sharing - Nothing moves. The 12 dynamic nodes remain on the inference side for the whole replay, so inference always holds its full peak capacity and Slurm always holds its 72 nodes. This is today's two-cluster arrangement, and every other policy is measured against it.

On-demand - The reactive rule of Section 4.2.5, run here with a fixed cooldown of 300 seconds: A dynamic node goes to Slurm only once a job is waiting with nowhere else to go, and returns once it has sat idle on the Slurm side for the cooldown or once the inference demand needs it back. The rule acts one move behind the load in both directions, and it lends a node only while the inference side keeps a margin of spare nodes (Section 5.3.1).

Fixed schedule - The dynamic pool moves to Slurm at 23:00 and back to Kubernetes at 06:00, every day (Section 4.2.5). It reads only a clock, with no forecast and no live measurement. It captures the quiet night but forgoes every daytime lull, and it cannot detect a public holiday or an unusually quiet afternoon.

Predictive - The forecasting policy of Section 4.2.5, run here at the 90th percentile of the Chronos-Bolt forecast plus a one-node safety floor. It forecasts once per hour from the last seven days of load and re-checks the split against that forecast every 15 minutes, so it can return nodes before the load arrives rather than after. It uses only data that a live service would have at that moment.

Oracle - The same 15-minute loop as the predictive policy, but fed the real future demand from the trace instead of a forecast. It cannot be built. It marks the upper bound of what any forecast could achieve with this mechanism, so the small gap between it and the predictive policy is the price of forecasting the load instead of knowing it.

5.3.1 Safety mechanisms for the inference side

The on-demand rule, the predictive policy, and the oracle share one safety net, which applies whatever the plan says. The fixed schedule does without the first two rules below, because it returns every node before the working day. Three rules keep lending from leaving the inference side short.

First, no policy lends the inference side's last spare nodes. The on-demand rule lends a node only while at least eight spare nodes remain (the margin M in Algorithm 1), and the demand check below keeps a reserve free on top of the current demand: two nodes for

the on-demand rule, one for the predictive policy and the oracle. This keeps the service from lending a node that inference is about to need.

Second, the demand check (rule 4 in Section 4.2.5): Every five minutes the service compares the real inference demand plus the reserve with the capacity on the inference side. If the demand is larger, it reclaims lent nodes at once, whatever the forecast had planned. For the predictive policy, this catches the hours in which the forecast was too low.

Third, reclaiming a node never kills a running batch job. When the service needs a lent node back, it first searches for an idle lent node and moves that one immediately. If every lent node is busy, it selects one and marks it for return, which has two effects: The Slurm scheduler assigns that node no further jobs, and the moment its current job finishes and the node is empty, the node re-images back to the inference side. A busy lent node is therefore drained, not interrupted. Its running job is allowed to finish, and only then does the node move. Nothing is preempted on either side, as RQ2 requires.

Two limits keep this draining short. A lent node accepts only jobs of up to four hours, enforced as a Slurm partition time limit, so a return waits at most four hours for a job to drain, and usually far less: The median batch job in the trace lasts about four minutes. A node that has just moved also cannot move again until the cooldown has passed, set to about two and a half times the move time, which prevents a node from oscillating between the clusters faster than either side can use it. While a busy node drains, inference falls back on its fixed nodes and on the reserve it held back. Under the predictive policy, 1,264 of the 203,691 requests in the six-day replay had to wait, and the longest wait was three minutes.

Algorithm 1 collects the rules of this section in one place, with the parameter values the simulation uses.

5.4 Forecast accuracy

The predictive policy can only be as good as the forecast it plans on, so we first measured how well Chronos-Bolt predicts the inference load. We evaluated it on the month-long request trace as a plain forecasting problem, with the negotiation service switched off. The month is divided into three-hour bins. From the eighth day on, we predicted each three-hour bin from the seven days before it and compared the prediction with what actually arrived. The policy's own forecast follows the same procedure on one-hour bins: At the start of each hour it forecasts that hour from the seven days before it, and it never sees the value it is about to predict.

We compared Chronos-Bolt with four simple rules that need no model: Repeat the last bin, repeat the same time yesterday, repeat the same time last week, and average the same time over the past week. Table 10 gives three error measures for each. The mean absolute error (MAE) is the average miss in requests per bin. The symmetric mean absolute percentage error (sMAPE) states that miss as a percentage, so it does not grow with the size of the load. The mean absolute scaled error (MASE) divides the error by that of the same-time-yesterday rule, so that rule reads 1.00 and any forecaster below it is more accurate.

Chronos-Bolt is the most accurate forecaster on every measure. It roughly halves the error of the best simple rule (MASE 0.49) and reduces the percentage error to 30%, against 48% or worse for the rules. It achieves this without ever seeing GWDG data in training: The forecast comes from the downloaded weights alone, yet it still outperforms every rule

Algorithm 1 Decision rules of the lending policies, with the parameter values of the simulation. All policies move a node with the same five phases and share the rules for lent nodes; the demand check protects the on-demand rule, the predictive policy, and the oracle. The oracle runs the predictive rule on the true demand instead of the forecast. Demand D and inference capacity are counted in GPUs, four to a node.

Parameters: $N_{\text{dyn}} = 12$ movable nodes; $N_{\text{fix}} = 6$ fixed inference nodes; cooldown $C = 300\,\text{s}$; limit on jobs placed on lent nodes $L = 4\,\text{h}$; reserve R of 2 nodes (on-demand) or 1 node (predictive); lending margin $M = 8$ nodes (on-demand); safety floor $F = 1$ node (predictive)

procedure MOVE(n, side) ▷ drain, reassign, reboot, join, verify: 120 s
  if side is Slurm **and** the inference side without n cannot hold the requests in flight **then**
    return ▷ never strand a running request
  end if
  carry n through the five phases
end procedure
procedure RECLAIM(k) ▷ bring k lent nodes back, idle ones first
  for k times **do**
    if a lent node n runs no job **then**
      MOVE(n, Kubernetes)
    else if a lent node n runs a job **then**
      mark n for return ▷ it takes no new job and moves once empty
    end if
  end for
end procedure

Lent nodes, all policies: a lent node accepts only jobs of at most L; a node marked for return accepts none and calls MOVE(n, Kubernetes) when its last job ends

Demand check, every 5 minutes (on-demand, predictive, oracle):

$k \leftarrow \lceil (D_{\text{now}} + 4R - \text{inference GPUs held or on their way back})/4 \rceil$
if $k > 0$ **then** RECLAIM(k)
end if

Predictive, every 15 minutes:

$\hat{D} \leftarrow$ p90 of the Chronos-Bolt forecast made at the start of the current hour from the 7 days before it (no new forecast every 15 minutes), in nodes
$T \leftarrow N_{\text{dyn}} - \min\big(N_{\text{dyn}}, \max(0, \lceil \hat{D} \rceil + F - N_{\text{fix}})\big)$ ▷ nodes to lend
$c \leftarrow$ nodes lent to Slurm or on their way there
if $c < T$ **then**
  MOVE(n, Slurm) for $T - c$ dynamic nodes n on the Kubernetes side
else if $c > T$ **then**
  take back $c - T$ lent nodes in turn: an idle one moves at once, a busy one is marked for return
end if

On-demand, when a Slurm job arrives while jobs are waiting:

if $\lfloor (\text{inference GPUs} - D_{\text{now}})/4 \rfloor - M \geq 1$ **then**
  MOVE(n, Slurm) for one dynamic node n on the Kubernetes side
end if

On-demand, when a lent node n falls idle:

after C, if n is still idle and no job is waiting: MOVE(n, Kubernetes)

Fixed schedule: at 23:00 lend all N_{dyn} dynamic nodes; at 06:00 call RECLAIM(N_{dyn})

built around the daily and weekly cycle of this series. This is the accuracy the predictive policy rests on, and Section 5.5 quantifies what it is worth in queue time.

Table 10: Accuracy of the inference-load forecast for three-hour bins over the month-long trace, on 192 three-hour bins (every bin from the eighth day on), each predicted from the seven days before it. MAE is the mean miss in requests per bin; sMAPE states the same miss as a symmetric percentage; MASE divides the error by that of the same-time-yesterday rule, so that rule reads 1.00 and lower is better. Chronos-Bolt roughly halves the error of the best simple rule with no training on GWDG data.

Forecaster	MAE (req/bin)	sMAPE	MASE
Last bin	3,191	82%	1.37
Same time yesterday	2,336	48%	1.00
Same time last week	3,213	82%	1.38
7-day average	2,735	59%	1.17
Chronos-Bolt	**1,132**	**30%**	**0.49**

5.5 Queue wait, inference service level, and idle capacity

Table 11 gives the main result. Without sharing, Slurm jobs wait 290 minutes on average, and the inference side, which holds all 18 of its nodes at all times, never delays a request. Every sharing policy shortens the Slurm queue. The fixed schedule, which knows only that nights are quiet, reduces the mean wait to 251 minutes, a 13% cut. The on-demand rule reaches 170 minutes, a 41% cut. The predictive policy reaches 166 minutes, a 43% cut, and the oracle, which knows the future exactly, reaches 165 minutes. The median matters more than the mean here, because it describes the job in the middle rather than the few very long waits: It falls from 186 minutes to 32 under the predictive policy, a cut of 83%. The typical job now waits about half an hour instead of three hours.

On the Slurm side the on-demand rule and the predictive policy therefore appear equivalent. They differ in what they cost the users of the inference service. Table 11 counts every request that had to wait for a free slot instead of starting at once. Under the on-demand rule, 5,222 requests waited, one in thirty-nine, for about two minutes each on average, and the worst waited almost seven minutes. Under the predictive policy, 1,264 requests waited, one in 161, for about a minute each on average, and the worst waited three minutes. Averaged over all requests, the added wait falls from three seconds to 0.4 seconds. The predictive policy therefore delays a quarter as many requests and causes a seventh of the total waiting time, at an equally short Slurm queue.

The on-demand rule lends a node only once it observes that the node is spare, and it reclaims one only once the inference side is already full, so it is always one step behind. When the morning load arrives, the nodes it needs are still on the Slurm side, and returning one costs two minutes plus however long the Slurm job on it still needs. The predictive policy has already returned them before the load arrives, so far fewer requests queue.

The forecast is accurate but not perfect. Its median prediction is off by about 1.1 nodes, and its 90th-percentile forecast, which should lie above the real demand in 90% of the hours, did so in 85%. The one-node safety floor and the demand check (Section 5.3.1) caught most of the remaining misses. The oracle shows what a better forecast would still gain: With a perfect forecast only 27 requests wait instead of 1,264, although the mean Slurm wait improves by barely a minute. A better forecast would therefore mainly reduce the delays on the inference side, not shorten the queue, and an ordinary forecast already collects most of the queue benefit.

Figure 17 shows where the gain originates. Panel (b) tracks the number of nodes

Table 11: Slurm queue wait and inference service level over the six-day window, on the same 90-node pool and the same real workloads. The inference columns count the requests, out of 203,691 in the window, that had to wait for a free slot instead of starting straight away; the mean wait is averaged over all requests, including those that started at once. No request is rejected; the service queues them. The on-demand rule and the predictive policy shorten the Slurm queue by about the same amount, but the predictive policy delays a quarter as many requests. The oracle is the same policy with a perfect forecast.

	Slurm queue wait (min)			Inference requests kept waiting			Node
Policy	mean	median	p95	how many	mean wait	worst	moves
No sharing	290	186	1305	0 (0.00%)	0 ms	0 s	0
Fixed schedule	251	115	1216	0 (0.00%)	0 ms	0 s	144
On-demand	170	41	968	5,222 (2.56%)	2,962 ms	406 s	232
Predictive (p90)	166	32	966	1,264 (0.62%)	406 ms	186 s	172
Oracle	165	32	964	27 (0.01%)	0.3 ms	7 s	150

available to Slurm over the week. The no-sharing line is flat at 72. The predictive and oracle lines rise and fall with the workload: Up to 84 nodes through the nights, then back down each morning, just before the inference load climbs again. The on-demand line moves less and later, because it lends only once a job is already blocked, and its margin of eight spare inference nodes limits how many it lends. Panel (c) shows the same behavior from the inference side: The predictive capacity line follows the shape of the demand and leaves Slurm everything the inference load does not require, while the on-demand line stays within a fixed band and returns capacity late.

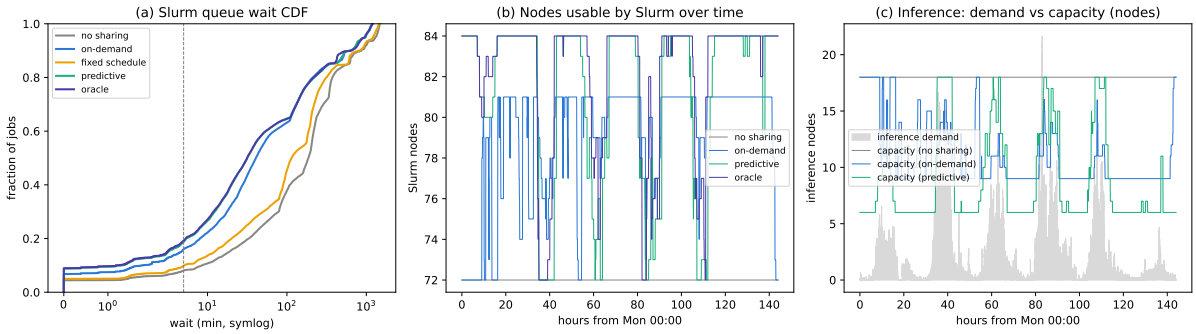


Figure 17: (a) Slurm queue wait as a cumulative curve (the fraction of jobs that wait less than a given time); every sharing policy shifts the curve left of the no-sharing baseline, the predictive policy furthest among the policies that can run in practice (all but the oracle). (b) Nodes Slurm can use over the week: flat without sharing, raised overnight and between daytime bursts under negotiation. (c) Inference demand (shaded) against the capacity each policy provides.

In short, the fixed schedule is safe but lends the least: It never delays a request, because it returns every node at six in the morning, but it forgoes every quiet hour during the day. The on-demand rule shortens the Slurm queue almost as far as the predictive policy, but it reacts late in both directions, and the inference side pays with four times as many delayed requests. The gap between the predictive policy and the oracle is the cost of an imperfect forecast, and it falls on the inference side rather than on the queue.

Idle capacity - The queue wait answers one half of RQ3. The other half asks whether sharing reduces the time nodes stand idle. Table 12 counts that idle time in node-hours, where one node-hour is one node running for one hour. Over the six days, the 90 nodes provide 12,960 node-hours in total (90 nodes × 144 hours). Without sharing, the 18 inference nodes stand idle for 2,324 of their 2,592 node-hours, or 90%, because they are sized for a peak the demand reaches only briefly. Every sharing policy lends part of that capacity to Slurm and cuts the idle time on the inference side: the fixed schedule by 23%, the on-demand rule by 44%, and the predictive policy by 57%, to 988 node-hours. Most of the remaining idle time comes from the six fixed inference nodes, which the design keeps on purpose: They alone provide 864 node-hours, more than three times the 268 node-hours the inference demand actually used.

Table 12: Idle capacity over the six-day window, in node-hours (one node for one hour). The 90-node fleet provides 12,960 node-hours in total (90 nodes × 144 hours), and the percentages in the "whole fleet" column are shares of that total. Idle time is counted per minute: inference capacity above the demand, Slurm nodes that run no job, and nodes in transition. The idle inference capacity depends on the sizing assumption of Section 6.1. The last two columns give the node-hours that movable nodes spent on the Slurm side and the part of them in which they ran a job. Lent nodes that ran no job were mostly blocked by a queued job that needed more than the four hours a lent node accepts; without backfill, the simulated scheduler lets no shorter job start ahead of it.

	Idle node-hours				Lent to Slurm	
Policy	inference	Slurm	moving	whole fleet	node-hours	running jobs
No sharing	2,324	684	0	3,008 (23.2%)	0	0
Fixed schedule	1,778	972	5	2,755 (21.3%)	541	262
On-demand	1,300	1,195	8	2,502 (19.3%)	1,016	585
Predictive (p90)	988	1,555	6	2,548 (19.7%)	1,330	597
Oracle	858	1,656	5	2,519 (19.4%)	1,460	631

Lending a node does not yet put it to work. Of the 1,330 node-hours the predictive policy lent, the nodes ran jobs for 597. For another 455, the job at the head of the queue needed more than the four hours a lent node accepts, and the simulated scheduler, which has no backfill (Section 4.4), lets no shorter job overtake it. A real Slurm, whose backfill scheduler starts shorter jobs out of order when they fit, would put much of that time to use, so this idle time reflects the simulator more than a lack of work. The remaining 278 node-hours fall in hours when the queue was empty: on the first Monday, when the replay starts from an idle fleet and a short queue, and in the two nights that follow, once the lent nodes had cleared the backlog. The on-demand rule lends less and leaves less of it idle, because it lends a node only when a job is waiting for it: Its lent nodes ran jobs for 585 of 1,016 node-hours. Both policies therefore move about the same amount of batch work onto lent nodes, and the on-demand rule ends slightly lower on idle capacity across the whole fleet, 19.3% against 19.7%, down from 23.2% without sharing. The advantage of the predictive policy lies in when it returns the nodes, not in how many hours it lends.

5.6 Transitions over one day

The numbers so far state what six days of sharing are worth in total. This section asks a narrower, practical question: At what time of day does the service move nodes, and how many does it move at once? Figure 18 answers it for the predictive policy on Wednesday 24 June, the middle day of the window. That day carries 38 of the week's 172 moves, 19 in each direction, more than the six-day average of about 29.

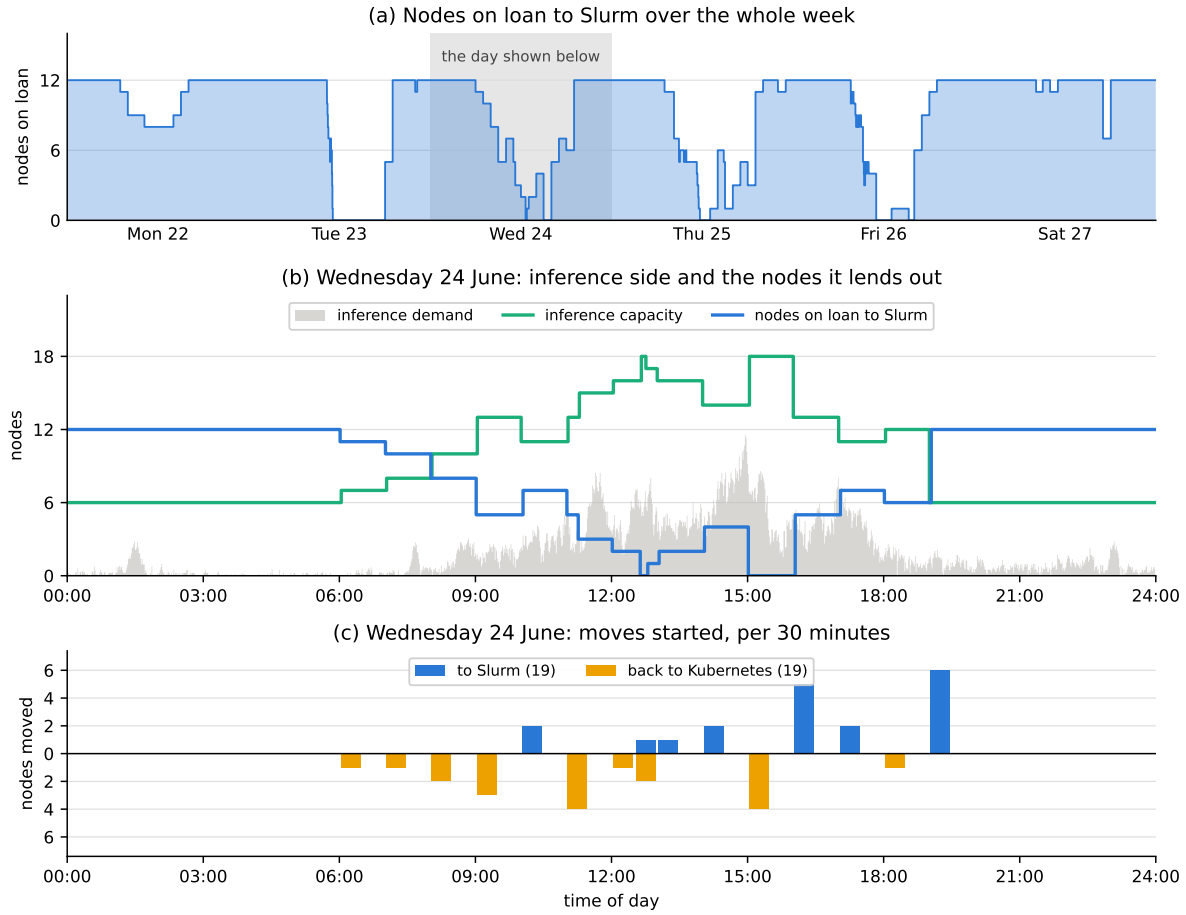


Figure 18: When the predictive policy moves nodes. (a) The movable nodes on loan to Slurm over the whole six-day window; the shaded day is the one below. (b) Wednesday 24 June: the inference demand (shaded), the capacity the inference side had, and the nodes it had lent to Slurm, all counted in nodes. (c) The moves of that day, counted in half-hour bins: Bars above the line are nodes going to Slurm, bars below are nodes coming back to Kubernetes.

Panel (b) shows the day as the service observed it. All twelve movable nodes were on the Slurm side overnight, because the inference load was low. They returned to Kubernetes through the morning, a few at a time, beginning before the working day did, so the inference side was already full when the load arrived. It held all eighteen of its nodes through the busy middle of the day. From late afternoon, as the load fell away again, the nodes returned to Slurm one by one, and by late evening the whole movable pool was lent out. Capacity stayed above demand all day.

Panel (c) shows the moves themselves. No node changes sides in bulk. Most half hours start one or two moves, so the Warewulf host re-images one or two machines at a time, and

a failed move costs the service one node rather than twelve. The cost of all this motion is small: At the two minutes per move measured in Section 5.1, the day's 38 moves amount to about 76 node-minutes in transit, well under one percent of the 288 node-hours the movable pool holds in a day.

Panel (a) shows that the same shape repeats. Every weekday follows the pattern of Wednesday: Lend in the evening, hold overnight, reclaim the nodes through the morning. Over the whole window, 76 of the 172 moves fall between 07:00 and 12:00, when the nodes return, and 84 between 14:00 and 22:00, when they are lent out again. Almost no node moves between 23:00 and 06:00, because the pool is already on the Slurm side and nothing is left to lend. The quiet days differ only in how long the nodes stay lent: On Saturday the inference load never rises far, so most of the pool stays with Slurm all day. The policy is stable: It makes about 2.4 moves per node per day, and they follow the daily rhythm of the users rather than oscillating.

5.7 Sensitivity analysis

Four settings could change the result: the size of the movable pool, the quantile the predictive policy provisions to, the safety floor, and the time one move costs. This section varies each of them in turn.

Movable pool size - Figure 19 varies how many nodes may move, from zero, where every policy coincides with the no-sharing baseline, up to the whole inference side, at a constant fleet of 90 nodes. The mean Slurm wait falls sharply as the first few nodes become movable and then flattens: Under the predictive policy it drops from 290 minutes at a fixed split to 220 minutes with three movable nodes, 193 with six, and 166 with twelve. Most of the benefit is therefore already present at six movable nodes, and beyond about a dozen it grows no further. The two policies diverge as the pool grows. The on-demand rule reaches its shortest wait, 170 minutes, at nine movable nodes and improves no further, while its inference cost climbs steadily, from one request in 330 delayed at three nodes to one in 39 at twelve. The predictive policy continues to improve up to twelve nodes and still serves 99.4% of requests there without any wait. A site therefore does not have to release much of its fleet to obtain most of the gain, and the more it does release, the more the forecast is worth.

Provisioning quantile - The forecast does not yield a single number but a range. Figure 20 varies where in that range the policy provisions, from the middle (p50) to the top (p90). Provisioning to the middle shortens the Slurm queue by a further twelve minutes, 154 instead of 166, and costs the inference side more than four times as many delayed requests, 5,639 instead of 1,264. Provisioning to the top is cautious and cheap: It gives up twelve minutes of queue wait (the queue is still 124 minutes shorter than without sharing), and it cuts the number of delayed requests by a factor of four. This is why p90 is used for every other result in this chapter. The same reasoning applies to any workload where provisioning too high costs a few lendable nodes and provisioning too low makes users wait: Provision high.

Safety floor - Provisioning high in the band is one form of caution; keeping spare nodes on top of the forecast, the safety floor, is another. Figure 21 varies the safety floor from zero to four spare nodes. The Slurm wait responds weakly, staying between 159 and 177 minutes across the whole range. The inference side responds more strongly, though not monotonically: With no floor 299 requests are delayed, with one node 1,264, with two 279,

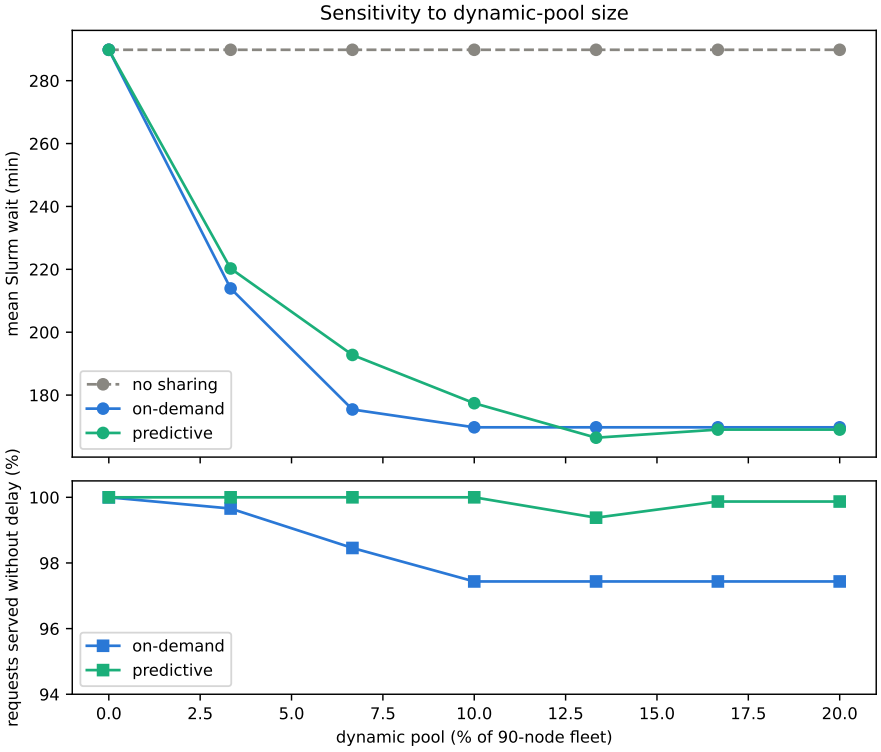


Figure 19: Mean Slurm wait (top) and the share of inference requests served without any wait (bottom), as the movable pool grows from zero to the whole inference side on a fixed 90-node fleet. Most of the queue benefit arrives with the first six movable nodes. The on-demand rule's inference cost climbs as the pool grows, whereas the predictive policy stays close to 100%.

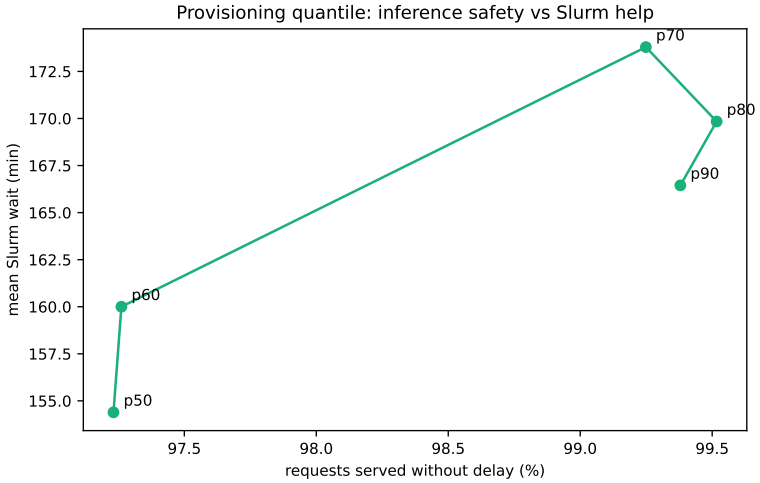


Figure 20: Mean Slurm wait against the share of inference requests served without any wait, as the predictive policy moves from the middle of the forecast range (p50) to the top (p90). Provisioning low shortens the Slurm queue slightly and delays far more requests; provisioning high is the cautious choice and costs almost nothing.

and from three nodes upwards none at all. The middle of the range is not ordered, because an extra spare node also shifts the moments at which nodes move and therefore changes which requests coincide with a transition. The two ends are unambiguous. Three spare nodes remove every delay for six minutes of additional mean Slurm wait, which makes spare nodes the cheapest protection in the design: A site that wants no delayed requests should add spare nodes first.

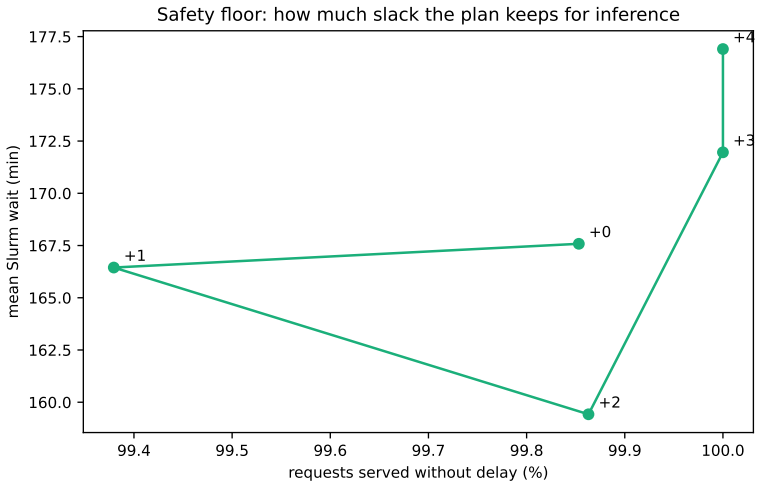


Figure 21: Mean Slurm wait against the share of inference requests served without any wait, as the safety floor grows from zero to four spare nodes on top of the forecast. Additional spare nodes remove the delayed requests entirely from three nodes upwards, while the Slurm queue changes by a few minutes.

Move time - Figure 22 repeats the on-demand rule, the predictive policy, and the oracle with the time for one move raised from 2 minutes to 60, and with the cooldown scaled to match. The Slurm waits change little, between 160 and 187 minutes across the whole sweep, which is expected: A node that needs an hour to arrive still relieves a queue that is hours long. The inference side is where slow moves are felt. The on-demand rule serves 97.4% of requests without any wait already at two minutes and never rises far above that, between 96.7% and 98.3% across the sweep, because it starts a move only once the load has arrived. The predictive policy starts at 99.4% and holds above 99% up to 15 minutes per move, since it begins its moves before the load arrives; only at 30 and 60 minutes does that lead time erode, to 98.1% and 96.7%, where even a plan made an hour ahead arrives too late. Fast moves therefore make reacting good enough, and slower moves make planning necessary, until a move takes about as long as the one-hour planning horizon. For this thesis the conclusion is the same either way: The faster the Warewulf provisioning path, the less the negotiation costs, and at the two minutes the prototype measured (Table 8) it costs very little.

5.8 Answers to the research questions

This section closes the chapter by returning to the four research questions of Section 4.1 and stating, for each one, what the evidence supports and how far it carries. One note on units first: Everything in this chapter counts *reserved* GPUs, whole cards allocated to a

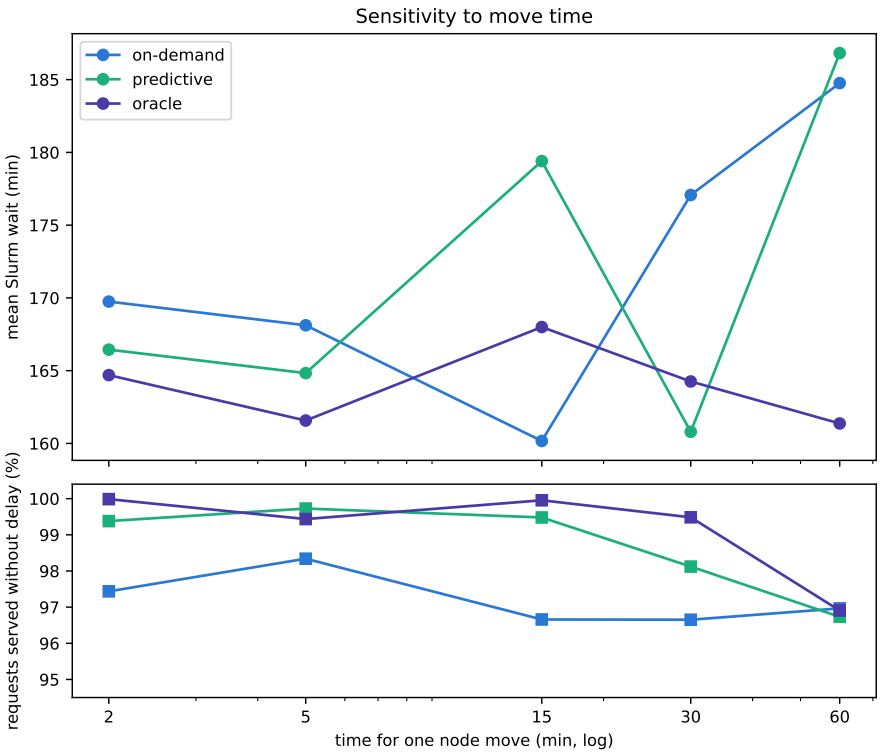


Figure 22: Mean Slurm wait (top) and the share of inference requests served without any wait (bottom), as one node move slows from 2 to 60 minutes. The Slurm queue changes little. The on-demand rule stays below 98.5% throughout, because it starts a move only once the load has arrived, while the predictive policy holds above 99% up to 15 minutes per move and converges with the reactive rule only at an hour.

job or to a serving instance, because whole cards are what both schedulers hand out. The utilization inside a card is a separate question, which this thesis does not measure.

RQ1: Can one Warewulf host provision the same node in either role? - Yes, by direct demonstration. The reference implementation keeps one image and one overlay per role on a single Warewulf host, and the same dynamic node boots as a Kubernetes worker one time and as a Slurm compute node the next, with the role determined solely by its assignment in Warewulf and nothing altered on the node itself. The confidence here is high, because the mechanism is not new: It is the same image-swap-and-reboot path Warewulf already follows for software upgrades, directed at a new purpose.

RQ2: Can the switch proceed automatically, without disrupting running work? - Yes for the automatic switch, with one part untested on the prototype. The daemon carries a node through all five phases with no manual step, and every move ends with a verified join or a logged timeout. The drains are built to wait for running work (the eviction budget on the Kubernetes side, the drain state on the Slurm side), but the testbed nodes carried no work when they moved, so the prototype never had to wait; the simulation models that wait from the real job runtimes instead. Draining under production load, where evicted inference pods must find room on busy neighbors, remains unmeasured.

RQ3: Does sharing help, and does prediction help beyond reacting? - Yes to both. Releasing about an eighth of the fleet for movement cut the median Slurm queue

wait by 83% and the mean by 43% against the fixed split. The typical job waited half an hour instead of three hours. That first finding requires no forecast at all: Even the fixed schedule, which knows only that nights are quiet, removes 13% of the mean wait. Sharing also cut idle time. The inference nodes, idle for 90% of their node-hours without sharing, stood idle 57% less under the predictive policy, and idle capacity across the whole fleet fell from 23% of all node-hours to under 20% under both the on-demand rule and the predictive policy (Table 12). A scheduler with backfill would likely lower it further, because much of the lent time that stayed idle waited behind a queued job too long for a lent node. Prediction pays off on the inference side. The on-demand rule reaches almost the same Slurm queue, but it reacts late, and 5,222 inference requests are delayed as a result, one in thirty-nine. The predictive policy delays 1,264, one in 161, and the longest wait falls from almost seven minutes to three: The same benefit for Slurm, with a quarter as many requests delayed. The oracle shows what a perfect forecast would add: The Slurm queue would stay almost the same, but only 27 inference requests would be delayed instead of 1,264. A better forecast would therefore mainly help the inference users, not the batch queue. These results come with the limitations listed in Section 6.1; the most important one is that the size of the inference service in the simulation is assumed, not measured.

RQ4: What does a transition cost, and which cooldown keeps the system stable? - A transition costs about two minutes, and one cooldown rule keeps the system stable. Over twenty prototype transitions the median took 98 seconds from Kubernetes to Slurm and 102 seconds back, with the reboot taking most of the time in both (Table 8); this sits alongside the under 90 seconds Decker and Kunkel report for a cold five-node cluster [DK25], on the same image-fetch and startup path. To that comes whatever the drain must wait for running work. On stability, a cooldown of five minutes, about two and a half times the move time, was enough to stop nodes from bouncing back and forth between the clusters; shorter cooldowns were not tested. Over the six simulated days, each policy made between 144 and 232 moves in total, which is about two to three moves per node per day. These moves follow the daily rhythm of the users, with nodes lent to Slurm in the evening and taken back in the morning (Figure 18). The predictive policy made 172 moves, fewer than the 232 of the on-demand rule. What remains unmeasured is the same as for RQ2: These timings come from a small testbed, not from production GPU nodes with large images.

Taken together, the four answers support the central question of this thesis: A single Warewulf-managed cluster can serve both systems on the same nodes, moved automatically. Slurm gains substantially against the fixed split, and the inference side gives up very little.

6 Discussion

This chapter interprets the results of the evaluation. Section 6.1 states how far those results carry before anything is generalized from them. Section 6.2 draws the practical lessons for computing centers that want to offer LLM services without purchasing a second GPU fleet. Section 6.3 examines how far the approach carries beyond Slurm, Kubernetes, and LLM serving. Section 6.4 names the larger problems the work touches but does not solve, and Section 6.5 collects what remains open, as a map for future research.

6.1 Limitations

A few simplifications limit how far these numbers carry, and the first is the most important. *The size of the inference service is an assumption, not a measurement.* Every request in the trace is real and is replayed at its real arrival time with its real duration, but the GPU share charged to each request is multiplied by a single constant, chosen so that the busiest moment of the week exactly fills the 18-node inference side. Without that step the observed traffic would leave the modeled inference side mostly empty, nothing would ever queue, every policy would score perfectly, and the comparison would show nothing. The constant is 7.35, so the experiment describes a service about seven times busier than the one the trace recorded. This affects how the numbers should be read. The comparison *between* policies is sound, because all five face the same workload on the same hardware. The absolute figures are not: "1,264 requests were delayed" is a statement about the modeled service, not a prediction about GWDG today. A separate check on the unscaled trace showed that at the volume actually observed the inference side never fills up at all, so today the whole movable pool could be lent out permanently. The experiment therefore models a service that has grown about sevenfold, not today's service.

The per-model GPU weights are likewise derived estimates from Table 9 rather than measured deployment state, and the model treats any inference node as able to serve any model, which a real deployment does not. Lent nodes accept only jobs of up to four hours. This limit is easy to set in Slurm, and it suits this trace, where the median job lasts only a few minutes. At a site where most jobs run for many hours, fewer jobs would fit on a lent node, and lending would gain less. The forecast was also not perfect: Its 90th-percentile value should lie above the real demand in 90% of the hours, but it did so in only 85%. In the remaining hours the forecast was too low, and the requests the predictive policy delayed come from these hours. The safety floor (one extra spare node) and the demand check (every five minutes against the real demand) caught most of these misses. The number of fixed Slurm nodes (72) was chosen to give a realistic baseline wait, so the wait times should be read as a fair comparison between the setups rather than as a prediction of GWDG's exact queue. The simulator schedules Slurm jobs first-come-first-served, with no backfill, which makes its waits an upper bound, and its move time is one fixed value per replay; the drain, reboot, and rejoin times behind it come from the prototype measurements of Section 5.1, taken on virtual machines rather than on production GPU nodes.

Last, the traces come from one site and one week, and the window holds only one weekend day, so the pattern the result rests on may differ at another site or in another season.

6.2 Implications for computing centers

A center that wants to offer LLM services today has two standard answers, and both are expensive. It can purchase a separate GPU fleet for the services, which then has to be sized for the midday peak and sits mostly idle at night and on weekends. Or it can host the services through the batch system, as SAIA does [Doo+25], which saves hardware but requires rebuilding service features inside a scheduler never intended for them. The results of this thesis support a third answer: one fleet, with a line between the two roles that follows the daily rhythm of the users. On the modeled workload, releasing about an eighth of the fleet for movement cut the median batch queue wait by 83%, while fewer than one inference request in 160 had to wait, for about a minute on average.

The ingredients behind this result are all ordinary. Stateless provisioning reduces a role change to an image assignment and a reboot, and many centers already operate Warewulf or a comparable system. Pre-listing the movable nodes in `slurm.conf` spares Slurm any configuration change when a node arrives. The control loop is one daemon of about 1,600 lines that invokes tools the site already operates. Even the forecasting, the component that sounds most like a research prototype, is a pretrained model applied unchanged, with no training on local data [Ans+24]. Running batch jobs and services on one fleet therefore needs no new middleware layer, only a small control loop on top of infrastructure most centers already operate.

Three findings matter most for a center planning such a system. First, the gains come early: The queue wait drops steeply as the first few nodes become movable and levels off around a dozen on this 90-node fleet (Section 5.7), so a site can start with a small movable pool and still collect most of the benefit. Second, caution is inexpensive: provisioning to the high end of the forecast band, and holding two or three spare nodes on top of it, cost Slurm almost nothing but reduced the number of delayed requests by a factor of four or more, so a site does not have to lend aggressively to benefit. Third, the side that lends the nodes pays almost nothing. This matters in practice: The team that runs the inference service will lend its nodes only if its own service does not suffer, and a policy that returns nodes before demand recovers makes that possible.

6.3 Beyond Slurm, Kubernetes, and LLMs

Reduced to its core, the approach requires four elements: nodes that boot their role from a central image, a batch scheduler that can drain a node and treats a node as present only while its node daemon runs, a service platform that moves work away from a drained node on its own, and a demand signal from each side. Nothing in that list is specific to Slurm or Kubernetes. HTCondor and PBS offer the same operations to drain a node and take it offline, and any orchestrator that can evict and reschedule pods could take the place of Kubernetes. The five-phase move and the two policies would carry over under different command names.

The pattern is also not tied to LLMs. The negotiation service never inspects a request; it observes a demand curve measured in GPUs and a queue of batch jobs. Any long-running service whose load follows the working day would create the same opportunity: interactive notebooks, remote visualization, rendering, continuous-integration runners on GPUs. The prediction component in particular is independent of the workload: Chronos-Bolt saw only a series of numbers and would forecast a JupyterHub load curve the same way.

Two conditions have to hold for the gains to transfer. The workloads must be complementary: One side must have quiet hours while the other carries a backlog, which

this thesis measured rather than assumed (Section 5.2), and which a site should verify in its own traces first. The batch queue must also contain enough short jobs, because a lent node has to come back quickly when the inference side needs it; on this trace most waiting jobs fit within the four-hour limit, whereas a queue of week-long jobs would need longer lending windows and would gain less. Scale is the remaining unknown: The moves are independent per node and the daemon is a single loop, so nothing in the mechanism obviously limits it to small clusters, but the testbed was small and larger scales remain untested.

6.4 Open problems

Security and credentials - Moving bare-metal nodes across a trust boundary, repeatedly and automatically, is the hardest open problem this work touches. The design rests on the companion security model [Mad26]: A node holds no credential at rest, receives a fresh single-use key on every move, and boots entirely from RAM, so that nothing survives a role change. That covers the node. It does not yet cover the daemon, which holds administrator rights on both schedulers and on the provisioning system at once and so makes the Warewulf host an even more valuable target than it already is. Scoping those rights, attesting that a node really booted the image it was assigned, and auditing every move are open engineering and research problems.

Multi-tenant fairness - Sharing changes who receives what, and the accounting has to follow. While a node serves Kubernetes, no Slurm account should be billed for it; that part is mechanical. The harder question is who should get the extra capacity: The donated night hours are additional capacity, and under fair-share scheduling they flow to whoever has jobs queued at night, which may favor the users who need the help least. Whether lent nodes should favor certain partitions, short jobs, or starved accounts is a policy question that the mechanism deliberately leaves open, and that a production deployment cannot leave open.

Energy-aware policies - This thesis treats an idle inference node as a node to lend. A center paying for power has a third option: Switch it off. Slurm already provides power-saving hooks (Section 2.3.9), and the negotiation loop could weigh three actions per node (hold, lend, sleep) against queue pressure, forecast demand, and electricity cost. This work did not study when switching a node off is worth more than lending it, and the answer will differ between a center with a backlogged queue and one without.

Monitoring integration - The prototype writes transcripts and CSV files, which suffices for experiments and not for operations. A production loop should export its state as live metrics, alert on failed joins, and, most importantly, monitor its own forecast: On the evaluated week the 90th-percentile forecast covered the truth in only 85% of the hours, and the safety net absorbed most of the difference. A deployed system should notice when its forecast starts missing more often, fall back to the on-demand rule on its own, and tell an operator why.

6.5 Research gaps

Four gaps remain between this thesis and a settled answer, and they define the future work.

- **Production validation.** The queue-wait results come from a simulator that replays real traces but simplifies the scheduler: no backfill, one fixed move time, one combined inference pool. The full loop, with real users and a real `slurmctld` with backfill, has not been exercised at production scale on either side.
- **A measured service model.** The GPU footprint of each model is derived from public model sizes and observed concurrency, because the platform's real replica counts were unavailable. Measuring the serving state directly, and forecasting per service or per token instead of per request, would replace the largest block of assumptions in this work with data.
- **Forecasting beyond zero-shot.** The pretrained model sufficed here, but its 90th-percentile forecast lay above the real demand in 85% instead of 90% of the hours, and this work tested neither fine-tuning, nor longer horizons, nor calendar effects such as holidays and semester breaks. How much a site-specific forecast improves the loop, and when zero-shot ceases to suffice, remains open.
- **Informing the schedulers.** The negotiation service treats both schedulers as black boxes, which makes it portable but also hides the lending plan from them. A Slurm that knows a node leaves at 06:00 could backfill it with jobs that finish in time; a Kubernetes that knows a node is about to return could pre-place pods. How much of the remaining gap to the oracle such co-design could close is unknown.

7 Conclusion

This thesis asked whether one Warewulf-managed cluster can serve both Kubernetes services and Slurm batch jobs on the same physical nodes, with the nodes moved between the two systems automatically as demand shifts. Within the limits stated in Section 6.1, the answer is yes. A single daemon on the provisioning host suffices to move the line between the two clusters, and on real GWDG workloads the batch queue became much shorter than under today's fixed split, at a small cost to the inference service.

7.1 Summary of contributions

For RQ1, the thesis treated a node's role as a property of its boot image and built a Warewulf setup with one image and one overlay per role, every movable node pre-listed in `slurm.conf`, and per-node keys delivered only at boot. The same node boots as a Kubernetes worker or as a Slurm compute node with no manual step on the node, through the mechanism Warewulf already uses for upgrades.

For RQ2, the thesis built the negotiation service: one daemon on the Warewulf host, about 1,600 lines of Bash around existing tools, with a five-phase move (drain, reassign, reboot, join, verify), a plain-text state file, and a systemd unit, and no software on any compute node. On the testbed it moved nodes unattended in both directions and logged failed moves for the operator instead of retrying them blindly. The drain is designed to protect running work on either side; the testbed exercised it only on idle nodes.

For RQ3, the thesis proposed two decision policies, an on-demand rule and a predictive policy built on a pretrained, zero-shot forecasting model, and compared them in a simulator that replays a month of inference requests and six days of Slurm jobs, aligned day for day. Sharing about an eighth of the fleet cut the median Slurm queue wait by 83% and the mean by 43%, and it more than halved the time the inference nodes stood idle. The predictive policy matched the reactive rule on queue time while delaying a quarter as many inference requests, and it came within one minute of the oracle, which reads the true future load from the trace.

For RQ4, the thesis built timing instrumentation into the daemon and varied the pool size, the provisioning quantile, the safety floor, and the move time in the simulator. A move costs about two minutes plus the drain, and a cooldown of about two and a half move times kept every policy stable at two to three moves per node per day. The value of prediction grows with the move time: Fast moves make reacting tolerable, and slow moves make forecasting necessary.

Beyond the four questions, the workload study stands on its own: Two production traces from the same days show that LLM inference swings almost sixtyfold over the day while batch GPU demand stays flat and backlogged. Other centers can run the same analysis on their own traces to see whether sharing would pay off for them.

7.2 Future work

The research gaps in Section 6.5 state what remains unknown; this section states what to build next.

- **Operate the loop in production.** Deploy the negotiation service on a real GPU partition with real users, with the on-demand rule as the safe starting policy, and

compare the measured queue waits against the simulator's predictions. This is the most valuable next step and the true test of RQ2 at scale.

- **Predict per service, not per pool.** The current demand curve merges all models into one series because the trace records no replica counts. Once the serving platform exports its instance data, the forecast can move per model, and the footprint table of Table 9 can be replaced by measurements.
- **Count tokens, not requests.** A request is a coarse unit of load: Two requests to the same model can differ in cost by orders of magnitude. Token-level accounting would sharpen both the demand curve and the forecast.
- **Stronger forecasts where needed.** Zero-shot Chronos-Bolt sufficed here, but its 90th-percentile forecast lay above the real demand in only 85% of the hours instead of 90%. Fine-tuning on site data, longer horizons, and calendar effects (holidays, semester breaks) are the natural extensions if a production loop shows the safety net triggering often.
- **Energy as a third option.** Extend the policy from hold-or-lend to hold, lend, or power down, and let electricity price and queue pressure compete for the idle hours.
- **More than two sides.** The state file and the daemon assume exactly two clusters. Several service platforms and several batch partitions negotiating over one pool is the general version of this problem, and nothing in the mechanism forbids it.

GPU hours are among the most expensive resources a research computing center operates, and part of them is lost not to missing demand but to a line drawn once through the hardware. The negotiation service moves that line with the daily rhythm of the users: The same cards that answer chat requests at noon can run batch jobs at midnight, at the cost of a reboot. The fixed division between an inference cluster and a batch cluster is not required. This thesis shows that it can be shifted automatically, without killing running work, with a large gain for batch users and a small cost to the inference service.

References

[Ans+24] Abdul Fatir Ansari et al. "Chronos: Learning the Language of Time Series". In: *Transactions on Machine Learning Research* (2024).

[Cal+15] Rodrigo N. Calheiros et al. "Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications' QoS". In: *IEEE Transactions on Cloud Computing* 3.4 (2015), pp. 449–458.

[Cha+23] Antony Chazapis et al. "Running Kubernetes Workloads on HPC". In: *Workshop on Converged Computing (WOCC), held in conjunction with ISC High Performance*. 2023.

[Das+24] Abhimanyu Das et al. "A Decoder-only Foundation Model for Time-series Forecasting". In: *Proceedings of the 41st International Conference on Machine Learning (ICML)*. 2024.

[DK25] Jonathan Decker and Julian Kunkel. "Ephemeral Kubernetes: dynamically deleting and recreating clusters using Warewulf'. In: *The Journal of Supercomputing* 81.12 (Oct. 2025), p. 1491. DOI: 10.1007/s11227-025-07668-y.

[DMK25] Jonathan Decker, Sören Metje, and Julian Kunkel. "Running Kubernetes Workloads on Rootless HPC Systems using Slurm". In: *Proceedings of the Sixteenth International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING 2025)*. IARIA, 2025.

[Doo+25] Ali Doosthosseini et al. *SAIA: A Seamless Slurm-Native Solution for HPC-Based Services*. July 2025. DOI: 10.21203/rs.3.rs-6648693/v1. URL: https://www.researchsquare.com/article/rs-6648693/v1.

[Doo+26] Ali Doosthosseini et al. "SAIA: a seamless Slurm-native solution for HPC-based services". In: *The Journal of Supercomputing* 82.403 (May 2026). DOI: 10.1007/s11227-026-08508-3.

[HA21] Rob J. Hyndman and George Athanasopoulos. *Forecasting: Principles and Practice*. 3rd. OTexts, 2021.

[HK06] Rob J. Hyndman and Anne B. Koehler. "Another look at measures of forecast accuracy". In: *International Journal of Forecasting* 22.4 (2006), pp. 679–688.

[KS18] Jitendra Kumar and Ashutosh Kumar Singh. "Workload prediction in cloud using artificial neural network and adaptive differential evolution". In: *Future Generation Computer Systems* 81 (2018), pp. 41–52.

[Kwo+23] Woosuk Kwon et al. "Efficient memory management for large language model serving with PagedAttention". In: *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. 2023.

[Lif95] David A. Lifka. "The ANL/IBM SP scheduling system". In: *Job Scheduling Strategies for Parallel Processing (JSSPP 1995)*. Vol. 949. Lecture Notes in Computer Science. Springer, 1995, pp. 295–303.

[Liu+18] Feng Liu et al. "Dynamically Negotiating Capacity Between On-demand and Batch Clusters". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE, 2018.

[Mad26] Shrinath Madde. *Security Analysis of Ephemeral Kubernetes in HPC Environments*. Seminar Report, Georg-August-Universität Göttingen, Institute of Computer Science. Supervisor: Jonathan Decker. Mar. 2026.

[Sha+20] Mohammad Shahrad et al. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 2020, pp. 205–218.

[Tra+25] Tim Trappen et al. "Automated Dynamic AI Inference Scaling on HPC-Infrastructure: Integrating Kubernetes, Slurm and vLLM". In: *Proceedings of Next-Gen Middleware for MLOps in Distributed Systems (MIND '25)*. Nashville, TN, USA: Association for Computing Machinery, Nov. 2025. arXiv: 2511.21413 [cs.DC]. URL: https://arxiv.org/abs/2511.21413.

[Ver+15] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. 2015.

[Wan+25] Yuxin Wang et al. "BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems". In: *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2025.

[Xia+26] Yuxing Xiang et al. "ServeGen: Workload Characterization and Generation of Large Language Model Serving in Production". In: *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2026.

[Xu+22] Minxian Xu et al. "esDNN: Deep Neural Network Based Multivariate Workload Prediction in Cloud Computing Environments". In: *ACM Transactions on Internet Technology* 22.3 (2022).

[YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing (JSSPP 2003)*. Vol. 2862. Lecture Notes in Computer Science. Springer, 2003, pp. 44–60.

[Zho+21] Naweiluo Zhou et al. "Container Orchestration on HPC Systems". In: *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*. arXiv:2012.08866. 2021.

A Transparency Report: Tools and Third-Party Contributions

A.1 Overview

- Writing
  - LaTeX, edited and compiled on Overleaf
- Images
  - Hand-drawn architecture and sequence diagrams: draw.io (diagrams.net)
- Diagrams
  - Script-generated architecture diagrams: the Python `diagrams` library
  - Script-generated plots and sequence charts: Python `matplotlib`
  - Sources for all script-generated figures are in `scripts/`
- Code
  - IDE: Visual Studio Code
  - AI-assisted code: written and optimized with the help of Chat AI (Claude, see below)
- AI
  - Anthropic Claude (Sonnet and Opus models), accessed via GWDG's internal Chat AI gateway

A.2 Per Section

- Section 1 Introduction
  - AI was used for brainstorming, for drafting or rewriting some passages (altogether about 10% of the section's text), and for proofreading and wording optimization
- Section 2 Background
  - AI was used for brainstorming, for drafting or rewriting some passages (altogether about 10% of the section's text), for proofreading and wording optimization, and for the scripts that generate the figures in this section
- Section 3 Related Work
  - AI was used for brainstorming, for drafting or rewriting some passages (altogether about 10% of the section's text), and for proofreading and wording optimization
- Section 4 Methodology

  - AI was used for brainstorming, for drafting or rewriting some passages (altogether about 10% of the section's text), and for proofreading and wording optimization
  - AI was also used to write, optimize, and restructure the code of the negotiation service and the scripts that generate the figures in this section

- Section 5 Evaluation
  - AI was used for brainstorming, for drafting or rewriting some passages (altogether about 10% of the section's text), and for proofreading and wording optimization
  - AI was also used to write, optimize, and restructure the simulation and analysis code, its test cases, and the scripts that generate the figures in this section

- Section 6 Discussion
  - AI was used for brainstorming, for drafting or rewriting some passages (altogether about 10% of the section's text), and for proofreading and wording optimization

- Section 7 Conclusion
  - AI was used for brainstorming, for drafting or rewriting some passages (altogether about 10% of the section's text), and for proofreading and wording optimization