

Master's Thesis

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

Lifecycle-Aware Scheduling and Resource Monitoring of ATLAS Workloads on the NHR-HPC Cluster EMMY

Prepared by
Ughur Mammadzada
from the Institute of Computer Science
at the II. Institute of Physics

First Supervisor: Prof. Dr. Arnulf Quadt
Second Supervisor: Prof. Dr. Julian Kunkel

Thesis number: II.Physik-UniGö-MSc-2025/08

Master's Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

30. September 2025

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

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

First Supervisor: Prof. Dr. Arnulf Quadt
Second Supervisor: Prof. Dr. Julian Kunkel

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 30. September 2025

Erklärung

nach §17(9) der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Universität Göttingen:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.

Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestanden Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, den 30. September 2025

(Ughur Mammadzada)

Abstract

As the university-based Tier-2 computing centres in Germany transition towards integration with national HPC facilities, the challenge arises of exploiting these resources opportunistically without disrupting primary production workloads. This thesis investigates methods for background job execution and monitoring within the established integration of the GoeGrid cluster with the NHR cluster EMMY. Building on the existing drone-based deployment, the work focused on enabling background jobs through Condor slots, isolation via cgroup-based controls and validation of scheduler behaviour under different activation policies.

A monitoring stack based on the ELK framework was implemented, offering visibility into network traffic, power consumption, node lifecycle states and PanDA queue assignments. These dashboards provided operational transparency and ensured that drones and queues behaved as expected in production-like settings. Controlled experiments showed that background jobs can successfully recover idle resources, but when activated prematurely they interfered with single-core foreground tasks, causing an efficiency loss. Restricting background jobs to draining phases mitigated interference while maintaining high utilisation.

In parallel, the drone management codebase was refactored to emphasise modularity, configurability and reproducibility. Although static drone lifetimes and limited error handling remain, the system provides a solid basis for future automation with COBaID/TARDIS.

Zusammenfassung

Mit der Umstellung der universitätsbasierten Tier-2-Rechenzentren in Deutschland auf die Integration mit nationalen HPC-Systemen stellt sich die Herausforderung, diese Ressourcen opportunistisch zu nutzen, ohne die primären Produktions-Workloads zu beeinträchtigen. Diese Arbeit untersucht Methoden zur Ausführung von Hintergrundjobs und zum Monitoring im Rahmen der etablierten Integration des GoeGrid-Clusters mit dem NHR-Cluster EMMY. Aufbauend auf der bestehenden, drohnenbasierten Infrastruktur lag der Schwerpunkt auf der Aktivierung von Hintergrundjobs über Condor-Slots, der Isolation mittels cgroup-Kontrollen sowie der Validierung des Scheduler-Verhaltens unter verschiedenen Aktivierungsstrategien.

Ein auf dem ELK-Framework basierender Monitoring-Stack wurde implementiert, der Einblicke in Netzwerkverkehr, Energieverbrauch, Zustände der Rechenknoten und PanDA-Queue-Zuweisungen bietet. Diese Dashboards ermöglichten Transparenz im Betrieb und stellten sicher, dass Drohnen und Queues unter produktionsähnlichen Bedingungen korrekt arbeiteten. Kontrollierte Experimente zeigten, dass Hintergrundjobs ungenutzte Ressourcen erfolgreich ausschöpfen können, jedoch bei zu früher Aktivierung mit einkernigen Vordergrundjobs interferierten und so einen Effizienzverlust verursachten. Die Beschränkung von Hintergrundjobs auf die Drain-Phase verhinderte Interferenzen und ermöglichte gleichzeitig eine hohe Gesamtauslastung.

Parallel dazu wurde der Code zur Verwaltung der Drohnen refaktoriert, um Modularität, Konfigurierbarkeit und Reproduzierbarkeit zu betonen. Obwohl statische Drohnenlaufzeiten und eingeschränktes Fehlerhandling bestehen bleiben, bietet das System eine solide Grundlage für zukünftige Automatisierung mit COBaID/TARDIS.

Contents

1	Introduction	1
1.1	High-Energy Physics Computing Context	1
1.1.1	CERN	2
1.1.2	ATLAS	4
1.1.3	WLCG	5
1.2	Local Infrastructure	7
1.2.1	GoeGrid	8
1.2.2	NHR EMMY	9
1.2.3	Transition from GoeGrid to EMMY	9
1.3	Software Ecosystem	10
1.3.1	Slurm	11
1.3.2	HTCondor	11
1.3.3	ELK Stack	12
1.3.4	cgroups	14
1.4	Problem Statement	14
2	Related Work	17
2.1	Distributed Infrastructures in HEP	17
2.2	Integration of HEP Workflows with HPC Systems	18
2.3	Virtualisation and Containerisation in Scientific Computing	18
2.4	Monitoring Frameworks for Distributed and HPC Systems	20
2.5	Scheduling Strategies and Opportunistic Computing	21
2.6	Summary and Open Gaps	22
3	Previous Work	25
3.1	Introduction	25
3.2	Resource Consumption Analysis	26
3.3	Monitoring Infrastructure Setup	28

4	Design and Methodology	31
4.1	Introduction	31
4.2	Monitoring Pipelines	31
4.2.1	Architecture and Design Rationale	32
4.2.2	Condor Reader	32
4.2.3	Network Reader	33
4.2.4	Integration with the ELK Stack	35
4.2.5	Summary	36
4.3	Background Jobs	36
4.3.1	Architecture and Drone Deployment	36
4.3.2	Slot Layout and Queue Separation	36
4.3.3	cgroup-Based CPU Management	38
4.3.4	Lifecycle Control and Experiment Modes	39
4.3.5	Validation	39
4.3.6	Summary	40
4.4	Codebase Refactoring	40
4.4.1	Methodology and Design Principles	40
4.4.2	Control via Command-Line Interface	41
4.4.3	Configuration File Reference	41
4.4.4	Cgroup Integration	43
4.4.5	Comparison with Original Implementation	43
4.4.6	Summary	44
5	Results	47
5.1	Monitoring Coverage	47
5.2	Impact of Background Jobs	49
5.3	Codebase Improvements	58
6	Discussion and Conclusion	61
6.1	Discussion	61
6.2	Limitations	62
6.3	Future Work	63
6.4	Conclusion	64
	Bibliography	70
	Appendix	73
	Description of ATLAS Job Types	73

List of Figures

1.1	The LHC, Geneva, at the border of Switzerland and France. Adapted from [1]. . .	3
1.2	Layout of the LHC and its four major experiments. ATLAS and CMS are located at the collision points with the highest energies, while ALICE and LHCb serve specialised physics programmes. Adapted from [1].	4
1.3	Schematic view of the ATLAS detector with its main subsystems: the inner detector, calorimeters and muon spectrometer. Adapted from [1,2].	5
1.4	Schematic view of the tiered structure of the WLCG. Data are collected at CERN (Tier-0), distributed to Tier-1 centres, then further processed and analysed at Tier-2 sites. Adapted from [3].	6
1.5	Schematic view of the connectivity of the WLCG. DE-KIT is a Tier 1 site located in Karlsruhe, Germany. Adapted from [3].	7
1.6	A drone with a lifetime of 2 days. New jobs are accepted only during the first day. The drone has 192 logical cores. "Draining" indicates the start of the period in which the drone is only running jobs that were accepted the previous day and have not finished yet.	11
1.7	The ELK Stack components: Elasticsearch for storage and search, Logstash for data ingestion and Kibana for visualisation.	13
3.1	Job share every month from 01.08.2023 to 31.07.2024 (top). The corresponding calculated bandwidth (middle). And the corresponding calculated memory (bottom). Calculated for 20000 cores. Processing type describes the type of the submitted job.	27
4.1	Component-level communication: Network Agents (NA) run on Worker Nodes (W-Node), Condor reader/observer (CO) and Power reader/observer (PO) run centrally on the Monitoring Node (M-Node). PO collects data from Dell Power/Energy reporting of each W-Node	33
4.2	Example of the network agent data pipeline, showing log file generation, Filebeat shipping, Logstash parsing, Elasticsearch indexing and Kibana visualisation. On the right the deployment location of each component is described.	34

5.1	Overview of the Kibana monitoring interface, combining metrics from Condor readers, network readers and power monitoring into interactive dashboards. . . .	48
5.2	Lifecycle state dashboard, showing nodes in Alive and Draining modes as reported by the scheduler integration.	49
5.3	Queue flag dashboard, displaying PanDA queue attributes attached to drones and confirming correct separation of jobs across different data-source queues.	49
5.4	Job mix of foreground jobs with (a) and without (b) background jobs on the node. Background jobs launched from the start. It can be seen that during this period there was a lot of User Jobs, which are not typical production jobs and do not have standard resource consumption pattern.	51
5.5	Full CPUs utilised on drones with no background jobs. Before draining the foreground jobs consume all available 192 CPUs. When the draining starts, CPU usage starts to gradually decline, due to jobs finishing and vacating the drone.	52
5.6	Full CPUs utilised on drones with background jobs. Before draining all 192 CPUs are utilised. When draining starts, foreground jobs gradually vacate the drone, as they finish, however the CPUs used still remains almost full thanks to the background jobs. Before draining it can be seen that foreground jobs have periodic declines, which is background job interference into foreground jobs. The interference range is ~ 0 -20 CPUs (~ 0 -10%), which is 5% on average observed CPU efficiency decline. Here, each "pillar" and each dip period in CPU usage before draining is a single background job. As there is more and more free CPUs due to foreground jobs finishing after draining, the pillars get narrower, the background jobs can use more CPUs and get faster.	52
5.7	CPU efficiency distribution of foreground jobs. Foreground jobs on nodes without background jobs (orange, dashed) and with background jobs (blue, solid) CPU efficiency is calculated as relation of CPU time to wall time multiplied by number of CPUs registered in PanDA.	53
5.8	Third experiment. Background jobs were activated only after Draining phase started. Although this eliminated background job interference into foreground jobs' CPU usage, the experiment revealed that foreground jobs do not occupy all CPUs before draining.	53
5.9	Job mix of foreground jobs with background jobs on the node for the third experiment. Job share significantly differs from the first 2 experiments (Fig. 5.4). This experiment has an overwhelming presence of predictable production jobs and minimal amount of user jobs.	54
5.10	Experiment 4. Nodes without background jobs. Foreground jobs, as in the third experiment do not consume all available CPUs before draining. The gaps are present, however they are not as significant.	54

5.11	Experiment 4. Nodes with background jobs launched from the start. The foreground jobs do not occupy all available CPUs before draining. Background jobs fill the gaps. However, background jobs still interfere with the foreground jobs and the downs in CPU usage of foreground jobs are more significant due to this factor.	55
5.12	Experiment 4. Nodes with background jobs launched starting from draining. Before draining foreground jobs do not occupy all available CPUs, even though the usage is close to maximum (192 CPUs). After draining background jobs use CPUs to maximum.	55
5.13	Experiment 4. Job mix of foreground jobs on nodes with background jobs launched from start (a) and with background jobs launched only during draining (b). Job share significantly differs from the first 2 experiments (Fig. 5.4). This experiment has an overwhelming presence of predictable production jobs and minimal amount of user jobs.	56
5.14	CPU efficiency distribution of foreground jobs on nodes with (blue, solid) and without (orange, dashed) background jobs. Background jobs launched from the start.	57
5.15	CPU efficiency distribution of foreground jobs on nodes with (blue, solid) and without (orange, dashed) background jobs. Background jobs launched only during draining.	57
1	The standard software workflow of the ATLAS experiment. Processing steps are represented by blue ovals, with output formats represented as red boxes. The various steps and data formats are described in the text. The background entering digitisation may be additional simulated HITS files, pre-digitised RDO files or specially processed RAW detector data. [4]	75

List of Tables

4.1	Comparison of original and refactored drone management implementations. . . .	44
-----	---	----

List of Abbreviations

ADC	ATLAS Distributed Computing	17
ALICE	A Large Ion Collider Experiment	2
API	Application Programming Interface	20
ARC-CE	Advanced Resource Connector – Computing Element	9
ATLAS	A Toroidal LHC ApparatuS	2
AWS	Amazon Web Services, Inc.	28
BigPanDA	PanDA’s Monitoring Extension	20
CERN	European Organization for Nuclear Research (Conseil Européen pour la Recherche Nucléaire)	2
CernVM	CERN Virtual Machine	18
ClassAd	Classified Advertisement	37
CLI	Command-Line Interface	41
CMS	Compact Muon Solenoid	2
COBalD	the Opportunistic Balancing Daemon	63
CPU	Central Processing Unit	8
CVMFS	CernVM File System	18
cgroup	control groups	10
DAGMan	Directed Acyclic Graph Manager (HTCondor component)	12
DIRAC	Distributed Infrastructure with Remote Agent Control	22
ELK	Elasticsearch, Logstash, Kibana	10
EMMY	High-Performance Computing Cluster EMMY (University of Göttingen)	8
GlideinWMS	Glidein Based WMS (Workload Management System)	17
GPU	Graphics Processing Unit	2
HEP	High-Energy Physics	1
HL-LHC	High Luminosity Large Hadron Collider	3
HPC	High-Performance Computing	1
HTTP	Hypertext Transfer Protocol	18
HTCondor	High Throughput Condor	10
I/O	Input/Output	14

IP	Internet Protocol	34
IPMI	Intelligent Platform Management Interface	32
JSON	JavaScript Object Notation	32
LHC	Large Hadron Collider	2
LHCb	Large Hadron Collider beauty experiment	2
MPI	Message Passing Interface	19
MSc	Master of Science	25
NCSA	National Center for Supercomputing Applications	18
NHR	National High-Performance Computing (Germany)	8
OS	Operating System	19
PBS	Portable Batch System	18
PanDA	Production and Distributed Analysis (ATLAS workload management system)	9
SIF	Singularity Image Format	43
SIGHUP	Signal Hang Up	37
SRE	Site Reliability Engineering	28
Slurm	Simple Linux Utility for Resource Management	8
TARDIS	Transparent Adaptive Resource Dynamic Integration System	63
UMA	Unified Monitoring Architecture	20
WAN	Wide Area Network	34
WLCG	Worldwide LHC Computing Grid	3
YAML	YAML Ain't Markup Language	41
startd	Start Daemon	36

Chapter 1

Introduction

1.1 High-Energy Physics Computing Context

High-Energy Physics (HEP) is a data-intensive science. Every advance in experimental capability, from higher beam energies to more sophisticated detectors, has gone hand in hand with increased demands on computing. The discipline has therefore developed a unique reliance on distributed and large-scale computational infrastructures, not only to process and store data but also to make it accessible to a global community of researchers. Understanding this context is essential to appreciate the role of local clusters and High-Performance Computing (HPC) systems in supporting modern HEP workflows.

At the core of HEP research are large international collaborations. Thousands of scientists contribute to the design, construction and operation of experiments and thousands more perform analyses on the resulting datasets. This scale is unmatched in most other sciences and requires a model in which computing resources are shared across institutional and national boundaries.

The growth in data volume is not linear but exponential. Detector technologies continually improve in resolution, precision and readout rate, leading to a rapid increase in both raw and derived data products. Trigger systems reduce data at the detector level by picking up only interesting events, but the output remains measured in petabytes per year for each major experiment. Moreover, much of the computational load arises not from the storage of raw data but from the repeated reprocessing, calibration and simulation tasks that are essential for extracting physics results. Each of these tasks multiplies the computational demand, creating a sustained pressure on available resources.

To cope with this, HEP has been at the forefront of adopting novel computing paradigms. The field pioneered the use of large distributed grids, federated storage systems and job scheduling across heterogeneous environments. These solutions have since influenced developments in cloud computing and large-scale data science. A defining characteristic of the HEP computing model

is its balance between central coordination and local autonomy: central facilities provide global services, while regional and institutional clusters contribute capacity according to their capabilities. This structure both spreads the load and provides resilience against local outages or bottlenecks.

In the future the challenge will increase. The upcoming High-Luminosity upgrade of the Large Hadron Collider (LHC) will increase data rates and analysis complexity by an order of magnitude. Meeting these requirements within realistic budget and energy constraints calls for both hardware and software innovation. Integration of HPC resources, efficient use of accelerators such as Graphics Processing Unit (GPU)s and opportunistic resource usage strategies are increasingly seen as critical elements. Monitoring systems that can provide fine-grained insight into resource utilisation and job behaviour are equally essential, as they allow sites to identify inefficiencies and optimise their operations.

Within this broader landscape, national and regional computing centres play a dual role. They contribute capacity to the global infrastructure while also supporting local research communities. Their effectiveness depends not only on raw performance but also on how well they integrate with the wider ecosystem. Transitioning such centres from traditional grid middleware to modern HPC environments is a delicate process that requires rethinking job scheduling, resource allocation and monitoring strategies. It is in this space that the present work is situated.

1.1.1 CERN

The European Organization for Nuclear Research (Conseil Européen pour la Recherche Nucléaire) (CERN) is the largest laboratory for particle physics in the world. Founded in 1954, it has grown into an international hub for both fundamental research and technological development. Located near Geneva on the French–Swiss border, CERN hosts a complex of accelerators and experimental facilities that serve a community of more than 12,000 scientists from over 100 countries [1]. Its mission is to advance knowledge of fundamental particles and their interactions, while also training the next generation of scientists and engineers and fostering international collaboration.

The core of the laboratory is the LHC, a circular proton–proton and heavy-ion accelerator with a circumference of 27 kilometres (Fig. 1.1). It is buried about 100 metres underground and connects to four major experiments: A Toroidal LHC ApparatuS (ATLAS), Compact Muon Solenoid (CMS), A Large Ion Collider Experiment (ALICE) and Large Hadron Collider beauty experiment (LHCb) (Fig. 1.2). Protons are accelerated in opposite directions and brought into collision at a design centre-of-mass energy of 14 TeV, making the LHC the most powerful accelerator ever built. The collider is supplied by a sequence of pre-accelerators, which progressively increase beam energy before injection into the main ring.

In addition to accelerators, CERN provides a wide range of support infrastructure. This includes experimental caverns for the detectors, extensive data centres and the initial computing resources for data recording and reconstruction. The Tier-0 centre at CERN is the entry point of all raw

data collected by the LHC experiments. Within seconds of being recorded, events are stored, reconstructed and distributed to Tier-1 centres worldwide through the Worldwide LHC Computing Grid (WLCG) [3]. This integration of accelerator operations with computing infrastructure is a defining feature of modern HEP and highlights the dual role of CERN as both a scientific and a technological institution.

CERN has also played a central role in the development of distributed computing and networking. The laboratory was an early adopter of large-scale grid computing and it was at CERN that the World Wide Web was originally invented to facilitate collaboration among physicists [5,6]. The same collaborative culture underlies today's computing model, where CERN acts as both the physical host of the accelerator and the global hub for data management and distribution. This combination of physics, engineering and computing places CERN at the core of the HEP research, which is the context of this thesis.



Figure 1.1: The LHC, Geneva, at the border of Switzerland and France. Adapted from [1].

The LHC is scheduled for a major upgrade to the High Luminosity Large Hadron Collider (HL-LHC), which aims to increase the integrated luminosity by roughly an order of magnitude beyond the original design, thereby enabling more precise measurements and greater sensitivity to rare processes [7]. The HL-LHC is expected to begin operation in 2030, increasing instantaneous luminosity via stronger focusing magnets, crab cavities and upgraded injector systems [7]. This enhancement places additional demands on the computing and data infrastructure: experiments will generate larger data volumes, higher pile-up (more simultaneous collisions per crossing)

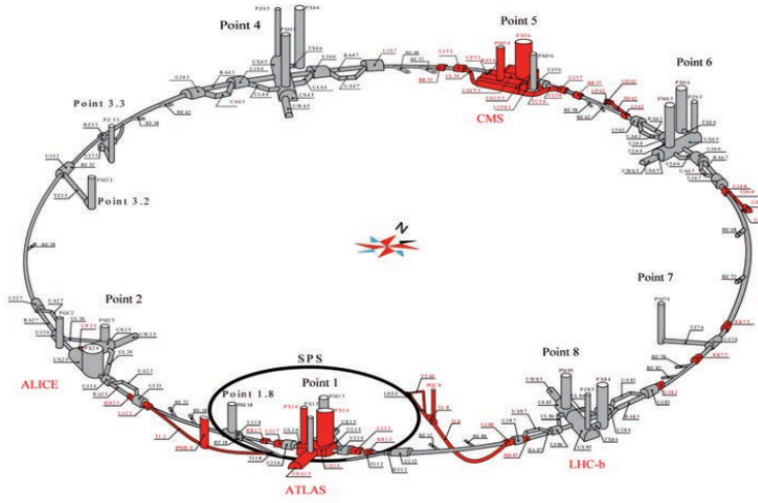


Figure 1.2: Layout of the LHC and its four major experiments. ATLAS and CMS are located at the collision points with the highest energies, while ALICE and LHCb serve specialised physics programmes. Adapted from [1].

and stricter real-time data processing requirements. As such, efficient integration of HPC and opportunistic computing resources becomes even more critical in the HL-LHC era.

1.1.2 ATLAS

ATLAS is one of the two general-purpose detectors at the LHC. It is designed to cover the widest possible range of physics processes, from precision measurements of Standard Model parameters to searches for new phenomena. The detector is located at Point 1 of the LHC ring and records collisions at a rate of up to 40 million interactions per second. A multi-level trigger system reduces this rate to a manageable level for permanent storage, selecting events of potential physics interest with high efficiency [2].

The ATLAS detector is cylindrical in design, measuring 44 metres in length, 25 metres in diameter and weighing about 7,000 tonnes. Its structure is organised in layers, each optimised for different types of particles (Fig. 1.3). The inner detector reconstructs charged particle tracks with high precision close to the interaction point. Surrounding it are electromagnetic and hadronic calorimeters, which measure the energy of electrons, photons and hadrons. The outermost system is the muon spectrometer, which uses toroidal magnetic fields to identify and measure muons. Together these systems provide nearly hermetic coverage, ensuring that most particles produced in a collision are detected and reconstructed.

The scientific programme of ATLAS is broad. It includes the study of electroweak interactions, the

strong force at high energies and the properties of heavy particles such as the top quark and the Higgs boson. ATLAS was central to the 2012 discovery of the Higgs boson [8], an achievement that confirmed the last missing element of the Standard Model. Since then, the experiment has focused on precision measurements of Higgs properties, searches for supersymmetry, extra dimensions, dark matter candidates and other phenomena beyond the Standard Model. Each of these research directions requires large datasets, complex statistical analyses and simulations for theory confirmations, which in turn depend on efficient computing and data management.

With more than 5,000 scientists from almost 180 institutions worldwide, ATLAS exemplifies the scale and collaborative character of modern HEP. Its success depends not only on the detector itself but also on the computing infrastructure that enables rapid data reconstruction, distribution and analysis. ATLAS therefore plays a central role in motivating developments in distributed computing and in shaping the requirements for systems such as the WLCG.

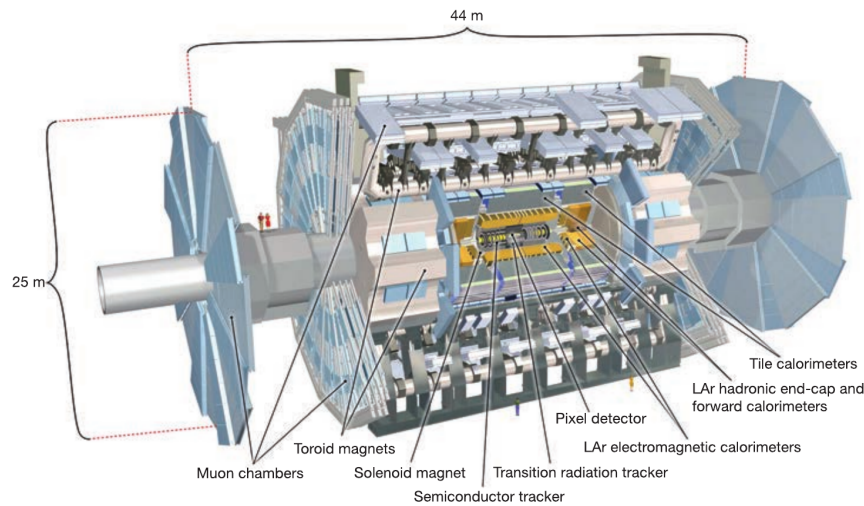


Figure 1.3: Schematic view of the ATLAS detector with its main subsystems: the inner detector, calorimeters and muon spectrometer. Adapted from [1,2].

1.1.3 WLCG

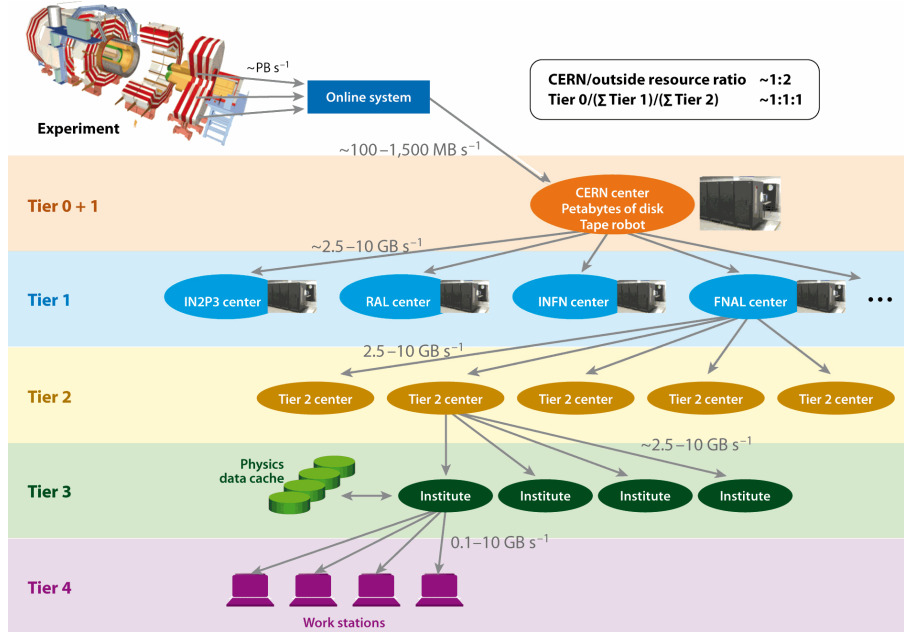
The WLCG is a globally distributed computing infrastructure developed to handle the big data volumes generated by the LHC experiments. No single site could provide the storage and processing capacity required. So a federated model was established in which resources from more than 170 computing centres in over 40 countries are combined into a coherent system [3]. The WLCG has become the largest scientific computing grid in operation, supporting tens of thousands of scientists worldwide.

The system is organised in a hierarchical tier structure (Fig. 1.4, 1.5). At the top is Tier-0, located at CERN, which is responsible for the initial recording and reconstruction of raw data. From

there, data are distributed to Tier-1 centres, large national facilities that provide long-term storage, reprocessing and high-availability services. Each Tier-1 site is connected to CERN and to several Tier-2 sites through dedicated high-bandwidth links. Tier-2 centres, often hosted by universities and regional computing facilities, provide resources for Monte Carlo simulation and user-level data analysis. In this way, the grid ensures that both central workflows and individual researcher analyses are supported in a balanced and scalable fashion.

The WLCG middleware handles job scheduling, data access, authentication and monitoring across heterogeneous sites. Over the years, the grid model has evolved to integrate new technologies such as virtualisation, cloud services and HPC resources. With the High-Luminosity LHC upgrade in the upcoming future, the WLCG is preparing for an order-of-magnitude increase in data volume and computational demand. Efforts are focused on more efficient resource usage, integration of accelerators and advanced monitoring frameworks.

For local Tier-2 sites such as GoeGrid, participation in the WLCG provides access to global workflows while also creating site-specific challenges. Each centre must ensure compatibility with grid middleware, provide reliable resources and adapt to evolving experiment requirements. The transition of such sites to the national HPC systems, while maintaining seamless integration into the WLCG, is one of the key motivations of this thesis.




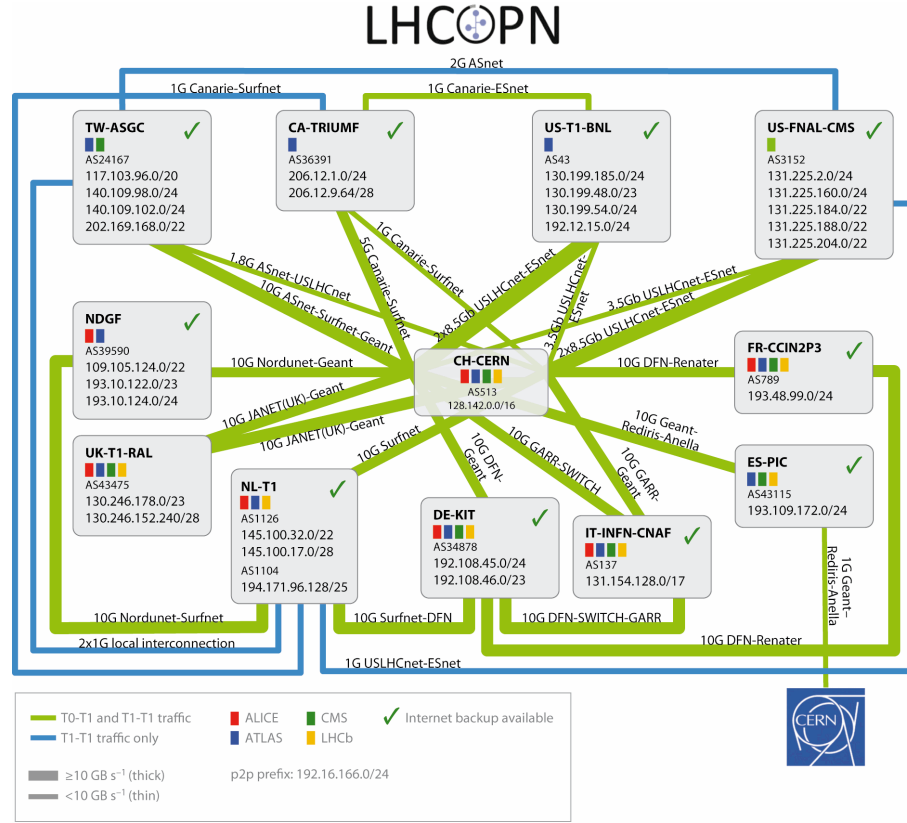
 Bird I. 2011.
Annu Rev. Nucl. Part. Sci. 61:99–118

Figure 1.4: Schematic view of the tiered structure of the WLCG. Data are collected at CERN (Tier-0), distributed to Tier-1 centres, then further processed and analysed at Tier-2 sites. Adapted from [3].



Bird I. 2011.
Annu Rev. Nucl. Part. Sci. 61:99–118

Figure 1.5: Schematic view of the connectivity of the WLCG. DE-KIT is a Tier 1 site located in Karlsruhe, Germany. Adapted from [3].

1.2 Local Infrastructure

The WLCG relies on the combined capacity of national and regional sites, which serve as Tier-1 or Tier-2 centres within the distributed infrastructure. These local facilities provide the bulk of the resources available for simulation and user-level analysis while also contributing to large-scale reprocessing campaigns. Their role is therefore essential, not only for the performance of the grid as a whole but also for supporting local research groups and training students and young scientists. Each site operates within the same global framework but adapts its configuration to the hardware and administrative environment available locally.

The University of Göttingen has been operating such a facility since 2008 in the form of GoeGrid, which acts as a Tier-2 and Tier-3 centre for the ATLAS experiment. Through its role in the WLCG, GoeGrid provides computing and storage resources that are directly accessible to the collaboration while at the same time serving the needs of the local particle physics group. The site enables

large-scale Monte Carlo production, user analyses and data processing tasks.

With the establishment of the National High-Performance Computing (Germany) (NHR) programme, new resources became available through modern HPC clusters. In this context, by the time current hardware at GoeGrid is phased out, no replacement is planned to be installed. Instead the computation power is planned to be provided from the resources available at the local NHR centre High-Performance Computing Cluster EMMY (University of Göttingen) (EMMY) [9]. Unlike traditional grid sites, EMMY is originally designed as a general-purpose HPC system, optimised for a wide range of scientific applications. Integrating such a system into the WLCG therefore requires adapting workflows, middleware and scheduling policies to a computing environment that differs significantly from classical grid clusters.

The transition from GoeGrid to EMMY is not simply a hardware upgrade but represents a structural change in how resources are provisioned and managed. Whereas grid middleware traditionally handled job submission and monitoring, HPC centres operate with batch scheduling systems such as Simple Linux Utility for Resource Management (Slurm), which impose different constraints on users and administrators. At the same time, the use of container technologies makes it possible to reproduce the software environment of grid worker nodes within HPC jobs. This hybrid approach allows experiments like ATLAS to continue running their workloads while taking advantage of modern HPC architectures. However, it also introduces new challenges in lifecycle management, monitoring and scheduling efficiency. Addressing these challenges is the main theme of this thesis.

1.2.1 GoeGrid

GoeGrid is a long-standing grid computing facility at the University of Göttingen that serves both as a Tier-2 and Tier-3 centre within the WLCG. As a Tier-2 site, it contributes resources to the ATLAS experiment for Monte Carlo production, reprocessing tasks and user-level analyses. At the same time, it acts as a Tier-3 resource for the local particle physics group, providing dedicated capacity for smaller-scale workflows, prototyping and training. This dual role illustrates the importance of local sites within the global infrastructure.

The hardware infrastructure of GoeGrid is based on a cluster environment with several thousand Central Processing Unit (CPU) cores and dedicated storage systems. Standard grid middleware ensured interoperability with the WLCG, allowing jobs to be submitted transparently from the ATLAS central workload management system. Storage elements were integrated into the global data federation, making datasets accessible to both local and remote users. In practice, GoeGrid supports thousands of jobs daily, depending on demand and availability.

Beyond providing raw capacity, GoeGrid also serves as a training environment for students and early-career researchers. Operating a Tier-2/Tier-3 centre exposes the local community to the complexities of distributed computing, including authentication, data management and workload scheduling. This knowledge transfer is valuable not only for physics analysis but also for broader

expertise in high-performance and distributed computing.

1.2.2 NHR EMMY

The NHR initiative in Germany provides researchers with access to modern computing infrastructure at several universities across the country. At the University of Göttingen, this role is fulfilled by the HPC clusters EMMY and GRETE.

EMMY was designed as a general-purpose HPC system rather than as a dedicated grid site. Its hardware configuration includes more than a hundred thousand CPU cores, high-speed interconnects and a hierarchical storage system combining parallel file systems with archival solutions. Unlike the grid clusters, which are usually composed of worker nodes managed by experiment-specific middleware, HPC systems like EMMY are optimised for a wide range of scientific applications, from computational fluid dynamics to machine learning. This diversity makes them highly powerful but also less tailored to the specific requirements of ATLAS workloads.

Resource management on EMMY is performed through the batch scheduling system Slurm, which enforces strict job lifetime and resource allocation policies. These policies ensure fair usage across disciplines but impose constraints that differ from GoeGrid. For example, the maximum run time of jobs and the priority-based scheduling mechanism can conflict with the needs of long-running or opportunistic physics workflows. Integrating ATLAS workloads into such an environment therefore requires additional layers of adaptation.

Despite these differences, EMMY offers significant advantages compared to the GoeGrid cluster. Its hardware is modern and energy-efficient, its interconnect enables high-bandwidth communication between nodes and its support infrastructure ensures long-term sustainability. By deploying containerised worker nodes within EMMY jobs, the software environment required by ATLAS can be reproduced reliably.

1.2.3 Transition from GoeGrid to EMMY

The transition from GoeGrid to EMMY marks a baseline change in how resources are provisioned for the ATLAS experiment at the University of Göttingen. While GoeGrid was operated as a conventional grid site, fully integrated with WLCG middleware and tailored to experiment requirements, EMMY is an HPC system designed to serve a broad scientific community. This difference meant that a direct migration of services was not possible. Instead, new strategies were required to adapt the ATLAS computing model to an HPC environment while preserving compatibility with the global grid infrastructure.

One of the key challenges was the change in workload management. On GoeGrid, jobs are submitted through the Production and Distributed Analysis (ATLAS workload management system) (PanDA) to GoeGrid middleware (Advanced Resource Connector – Computing Element

(ARC-CE)) as pilot jobs. Those pilot jobs are then run on the GoeGrid worker nodes and the pilot job subsequently pulls from PanDA the actual job payload.

In contrast, EMMY relies on the Slurm batch scheduler, which enforces strict run time limits, job prioritisation and fair-share policies. This means ATLAS jobs would need to compete with other jobs for the resources. Furthermore, EMMY does not have the software stack required for the ATLAS jobs. EMMY has its own update schedule, which might conflict with update schedules of ATLAS sites. These constraints necessitate an additional layer between ATLAS and the HPC system.

The adopted solution is to run containerised worker nodes (drones) inside Slurm jobs, effectively transforming HPC resources into virtual grid nodes for the duration of their allocation. This allows ATLAS software and middleware to operate in a familiar environment, while respecting the policies of the underlying HPC system.

Another issue is the difference in job lifecycle. Grid jobs can, in principle, run indefinitely as long as resources are available, whereas on EMMY jobs are bound by maximum wall time limits. If the virtual worker job can run for 7 days, accepting ATLAS jobs on that worker node on the last day would mean they most likely will be lost due to the time limit of the host Slurm job. This required adjustments in scheduling strategies, particularly to minimise job loss. However, this introduced considerable amount of CPU hours not utilised towards the end of the drone lifecycle (Fig. 1.6). To make use of idle resources towards the end of container lifetimes motivated the development of lifecycle-aware background jobs, which exploit otherwise wasted CPU hours without interfering with primary workloads.

Overall, the move from GoeGrid to EMMY demonstrates both the opportunities and the challenges of integrating HPC systems into the WLCG. It enables the use of modern, sustainable infrastructure while requiring solutions for workload adaptation, monitoring and scheduling efficiency. These solutions form the central focus of this thesis.

1.3 Software Ecosystem

The successful integration of HEP workloads into HPC systems depends not only on hardware but also on the software ecosystem that manages job scheduling, resource usage and monitoring. In the case of the transition from GoeGrid to EMMY, several key technologies play a central role. These include the Slurm batch scheduler, which governs access to HPC resources; High Throughput Condor (HTCondor), which is used by ATLAS to manage and distribute workloads; the Elasticsearch, Logstash, Kibana (ELK) Stack, which provides monitoring and visualisation capabilities; and control groups (cgroup), which enforces resource isolation at the operating system level. Together, these tools form the foundation on which lifecycle-aware scheduling and monitoring strategies can be built.

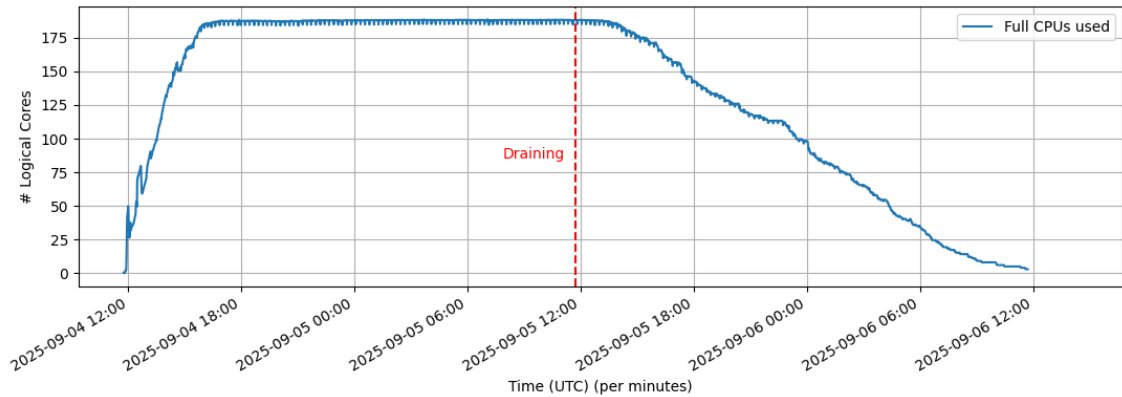


Figure 1.6: A drone with a lifetime of 2 days. New jobs are accepted only during the first day. The drone has 192 logical cores. "Draining" indicates the start of the period in which the drone is only running jobs that were accepted the previous day and have not finished yet.

1.3.1 Slurm

Slurm is the workload manager used on the EMMY HPC cluster. It is an open-source batch scheduling system widely adopted in HPC environments due to its scalability and flexibility. Slurm is responsible for allocating compute resources, enforcing fair usage policies and managing job queues. Users submit jobs to Slurm by specifying required resources such as CPU cores, memory and wall time. The scheduler then determines when and where each job will run, taking into account priorities, limits and available capacity.

Slurm enforces controlled access to a single HPC facility. In practice, this means that jobs on EMMY are subject to maximum run time limits and are scheduled according to fair-share policies that balance demand across disciplines. These constraints can be restrictive for physics workloads, which often consist of many independent tasks that must run reliably over long periods. Nevertheless, the predictability and efficiency of Slurm make it a robust platform for resource management at scale.

For the integration of ATLAS workloads, Slurm acts as the gateway between the WLCG and the HPC system. Containerised worker nodes are submitted to Slurm as standard batch jobs, effectively transforming allocated resources into temporary grid worker nodes. This books a number of EMMY nodes for a defined period of time (7 days) to be used to run ATLAS jobs exclusively. When the drone is near the end of its lifetime, a new drone can be launched via Slurm again.

1.3.2 HTCondor

HTCondor is a specialised workload management system designed for high-throughput computing. Unlike Slurm, which focuses on efficiently scheduling jobs within a single HPC facility,

HTCondor is optimised for managing large numbers of independent tasks across heterogeneous and geographically distributed resources. It is widely used in scientific computing and serves as the backbone of workload management for the ATLAS experiment within the WLCG.

The strength of HTCondor lies in its ability to handle opportunistic resources. Jobs are represented as independent tasks that can be dispatched to any available worker node in the system, with built-in mechanisms for job checkpointing, prioritisation and fault tolerance. The matchmaking process between jobs and resources is highly flexible, allowing workloads to be distributed across a wide range of environments, from dedicated clusters to opportunistic desktop resources. For ATLAS, this flexibility is essential, as it ensures that computational campaigns can scale across hundreds of sites with varying levels of capacity and reliability.

HTCondor is also well suited for workflow management. Its Directed Acyclic Graph Manager (HTCondor component) (DAGMan) allows users to express dependencies between tasks, ensuring that complex analysis pipelines and simulation campaigns are executed in the correct order. This capability is critical for the large-scale production of simulated data, which often involves multiple stages of generation, reconstruction and validation.

In the context of the transition from GoeGrid to EMMY, HTCondor continues to serve as the workload manager on the grid side. Jobs destined for Göttingen are first processed by PanDA, which sends the pilots to ARC-CEs. Those use HTCondor to assign tasks to available worker nodes, including drones. On EMMY, these jobs run within containerised environments submitted through Slurm. This dual-layer architecture, HTCondor at the grid level and Slurm at the HPC level, illustrates the challenge of integrating HEP workloads into HPC systems. Ensuring seamless interaction between these two schedulers is a key requirement for maintaining Göttingen's contribution to the WLCG.

1.3.3 ELK Stack

The ELK Stack is a collection of open-source tools for storing, processing and visualising logs and monitoring data. It consists of three main components: Elasticsearch, Logstash and Kibana. Together they provide a flexible framework for centralised logging, real-time analytics and interactive visualisation, which has made the stack widely adopted in both industry and research environments.

Elasticsearch is a distributed search and analytics engine that stores data in a document-oriented format. Its scalability and indexing capabilities make it suitable for handling large volumes of semi-structured data such as system logs, job records and performance metrics. Queries can be performed efficiently across millions of entries, allowing administrators and users to identify patterns and anomalies in resource usage.

Logstash serves as the data processing pipeline. It ingests information from diverse sources, including system logs, application outputs and monitoring agents, applies filtering and transformation

and forwards the results into Elasticsearch. Its plugin-based architecture allows the integration of many data formats and sources without requiring extensive custom development. This flexibility is particularly valuable in heterogeneous computing environments where multiple logging formats coexist.

Kibana provides the user interface for data exploration and visualisation. It offers dashboards, plotting tools and search functionality, enabling users to interact with data stored in Elasticsearch. Administrators can monitor cluster performance in real time, while researchers can analyse trends such as job efficiency or failure patterns. The combination of search, analytics and visualisation tools makes Kibana central to the usability of the ELK Stack.

For high-performance and distributed computing, the ELK Stack addresses the challenge of managing large amounts of operational data. By consolidating logs and metrics from multiple sources, it supports both debugging and long-term optimisation. In the context of integrating ATLAS workloads on EMMY, ELK provides the foundation for building custom monitoring pipelines that track drones, job performance and resource utilisation.

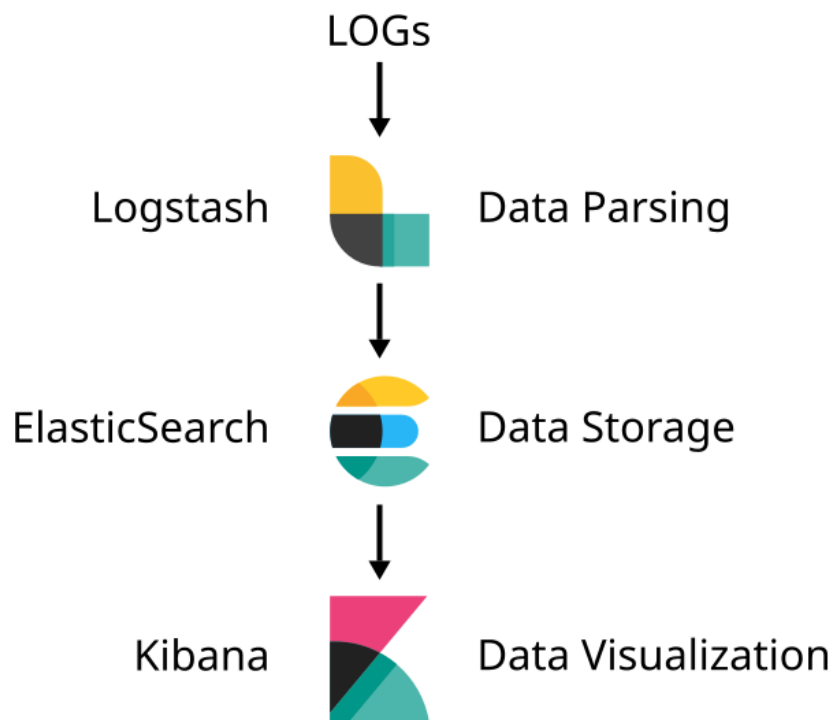


Figure 1.7: The ELK Stack components: Elasticsearch for storage and search, Logstash for data ingestion and Kibana for visualisation.

1.3.4 cgroups

On system level, cgroup are a Linux kernel feature that enables fine-grained control of system resources among groups of processes. Introduced in kernel version 2.6, cgroup allows administrators to limit, prioritise and account for resource usage such as CPU time, memory and Input/Output (I/O) bandwidth. They are widely used in container runtimes, cloud platforms and HPC environments where strict isolation between workloads is required.

At their core, cgroup organises processes into hierarchical groups, with each group subject to specific resource policies. For example, CPU shares can be allocated to ensure that no single job monopolises processing power or memory limits can be applied to prevent runaway processes from destabilising a system. This capability is essential in multi-tenant environments, where diverse applications share the same physical infrastructure.

In the context of HEP computing and this thesis, cgroup is particularly useful for separating opportunistic background jobs from primary production workloads. Background jobs can be constrained so that they only consume idle resources and yield immediately when higher-priority tasks require them. This ensures that opportunistic use of resources does not interfere with the performance or reliability of the main experiment workflows. Without such mechanisms, running additional jobs during idle periods would risk degrading the throughput of critical tasks.

On modern HPC systems such as EMMY, cgroup is integrated with the Slurm workload manager, which automatically applies resource limits based on user job specifications. In addition, container runtimes used to encapsulate ATLAS worker nodes rely on cgroup to enforce isolation between jobs. This combination makes cgroup a central element in lifecycle-aware scheduling strategies, providing the technical basis for safely exploiting idle CPU cycles without compromising primary workloads.

1.4 Problem Statement

The transition from GoeGrid to the NHR cluster EMMY places the Göttingen particle physics group at the intersection of two distinct computing environments. On the one hand, the WLCG has developed around grid sites, where resources are dedicated to experiment workloads and controlled by grid middleware. On the other hand, modern HPC centres such as EMMY operate with batch schedulers, strict run time limits and fair-share policies designed to serve a broad range of scientific disciplines. Bridging these two is essential for maintaining the contribution of the Göttingen site to ATLAS while aligning with the national HPC strategy.

Several challenges arise in this context. First, ATLAS workflows are not naturally adapted to the constraints of HPC systems. Many jobs are long-running, rely on opportunistic scheduling, and require software environments that differ from those used by other scientific fields. These characteristics conflict with the policies of HPC clusters, where run time and resource allocations

are tightly controlled. Without adaptation, physics workloads cannot be executed reliably on such systems.

Second, the lifecycle of jobs on EMMY introduces inefficiencies. Containerised worker nodes have a fixed lifetime imposed by the scheduler. To minimise job losses, production workloads are stopped well before the end of a container's lifetime. This results in idle CPU hours that have already been allocated but remain unused. Recovering this capacity in a way that does not interfere with primary workloads represents a significant optimisation opportunity.

Third, monitoring and operational transparency are limited in heterogeneous environments. Traditional grid monitoring tools are not fully compatible with limited access to the HPC system. Without a comprehensive monitoring framework, inefficiencies may remain undetected and opportunities for optimisation cannot be systematically exploited.

Addressing these challenges requires a combination of lifecycle-aware scheduling, fine-grained resource isolation and robust monitoring. The goal of this thesis is to design and implement methods that enable the efficient use of HPC resources for ATLAS workloads, with particular emphasis on background jobs that exploit idle capacity and on monitoring pipelines that provide detailed insights into system behaviour. In this way, the work contributes to the sustainable integration of national HPC resources into the global WLCG infrastructure.

Chapter 2

Related Work

2.1 Distributed Infrastructures in HEP

The architecture and operation of large-scale computing infrastructures for HEP have evolved significantly, with the WLCG being among the most important examples. Several studies have examined how the distributed grid model has been implemented, the operational challenges it has faced, and how to scale and monitor such large systems.

One relevant work is Lee et al. [10], which describes how ATLAS Distributed Computing (ADC) provides offline data processing, dataset management and workflow management across many sites. This study outlines the complexity of running distributed workloads reliably, dealing with failures and ensuring service availability across diverse hardware and administrative domains.

Operational monitoring has also been a major research topic. Sedov et al. [11] discuss how ATLAS developed dashboards and accounting systems to monitor site performance, job statuses and resource efficiency across grid sites. These tools were essential in detecting misbehaving sites, automating blacklisting and improving reliability at the site level.

A broader perspective is provided by Turilli et al. [12], who review pilot-job systems used in distributed infrastructures, including PanDA and Glidein Based WMS (Workload Management System) (GlideinWMS), which are core to the WLCG and the Open Science Grid. They identify common abstractions, such as late binding and resource placeholders and analyse how these systems cope with heterogeneous resource pools. Their work shows that pilot-job systems are a critical enabling technology for large-scale distributed infrastructures.

In addition, Stagni et al. [13] report on strategies to integrate LHCb workloads onto the Marconi-A2 supercomputer, highlighting adaptations in memory usage, multi-process execution models and submission policies required to match HPC constraints. This illustrates how distributed models developed for the WLCG are beginning to converge with HPC practices.

2.2 Integration of HEP Workflows with HPC Systems

As data volumes and computational demands from the LHC experiments continue to rise, significant effort has been devoted to exploring the use of HPC resources in addition to traditional grid infrastructures. The WLCG remains the backbone of distributed computing in HEP, but HPC centres offer attractive capabilities, including cutting-edge hardware, energy-efficient designs and high-bandwidth interconnects. Integrating the two models, however, is not straightforward, and several projects have investigated technical and organisational solutions to bridge the gap.

One of the earliest efforts was the integration of ATLAS production workloads on leadership-class supercomputers in the United States. The Blue Waters supercomputer at the National Center for Supercomputing Applications (NCSA) was used as a testbed for running ATLAS jobs within an HPC environment [14]. These studies demonstrated that with appropriate pilot-job models and adaptations, it is possible to map high-throughput tasks onto HPC batch systems without significant loss in efficiency.

Another area of progress has been the development of pilot frameworks that allow grid-like execution models to operate on HPC clusters. The PanDA workload management system used by ATLAS has been extended with the Harvester service, which provides a modular interface to external batch systems [15]. This allows PanDA pilots to run on HPC resources through local schedulers such as Slurm or Portable Batch System (PBS), dynamically provisioning worker nodes inside these environments. Similar strategies have been explored in CMS and ALICE, with varying degrees of success depending on the local HPC policies and access restrictions.

HPC systems typically enforce short maximum wall times, restrict outbound networking and have strict software validation policies. These factors complicate the execution of long-running or opportunistic high-throughput tasks. Moreover, data movement between HPC centres and WLCG storage federations is often limited by policy or bandwidth constraints. Addressing these issues requires continuous collaboration between HEP computing experts, HPC administrators and funding agencies. The experience gained so far indicates that hybrid models, where HPC centres contribute opportunistic or campaign-style resources to the WLCG, are both technically feasible and strategically valuable.

2.3 Virtualisation and Containerisation in Scientific Computing

A recurring theme in HEP computing is the need to reproduce complex software stacks across heterogeneous sites without sacrificing performance or maintainability. Early work addressed this with full virtualisation. The CERN Virtual Machine (CernVM) project provided a minimal virtual appliance tailored to experiment software, aiming to decouple user environments from site specifics while keeping operational overhead low [16]. In parallel, the CernVM File System (CVMFS) introduced a content-addressable, Hypertext Transfer Protocol (HTTP) based, globally cached,

read-only filesystem to distribute experiment software and conditions data efficiently to worldwide worker nodes [17]. Follow-up efforts reduced image footprint and provisioning cost (Micro-CernVM) [18] and evolved CVMFS towards more responsive, container-oriented and extreme-scale use cases [19–21].

As sites diversified (grid, HPC, cloud), Operating System (OS) level virtualisation became the dominant mechanism to deliver reproducible environments with near-native performance. Singularity/Apptainer, designed for unprivileged container execution on multi-tenant systems, demonstrated minimal overhead for typical scientific workloads and provided the security model required by HPC centres [22]. Systematic evaluations within HEP confirmed that containers are a viable alternative to full virtualisation for production codes, with performance close to bare metal for CPU, I/O and Message Passing Interface (MPI) bound tasks [23].

Experiments have since embedded containerisation into their operations. In ALICE, container management has been used to encapsulate grid services and streamline operational deployment across sites, reducing configuration drift and simplifying upgrades [24]. CMS has reported several production studies that leverage Singularity to execute workloads at HPC centres with constrained networking, using custom images and controlled data staging to satisfy site security policies [25–27]. These reports emphasise practical aspects that matter to operations: keeping images small, pre-placing dependencies on shared filesystems and aligning run time configurations with local schedulers.

CVMFS remains the backbone for shipping experiment software, even into containerised contexts, because it minimises wide-area distribution of large images: the base container can stay slim, while the experiment stack streams on demand from CVMFS caches [19]. Recent studies extend CVMFS to supercomputing environments and subset repositories, enabling curated distributions that reduce metadata pressure and start-up latency on leadership systems [21].

Despite the maturity of these tools, several challenges recur in the literature. First, container start-up and image distribution can become bottlenecks at scale if images are large or not cached effectively; strategies include CVMFS backed software, node-local caching and layered images aligned with site modules [19,27]. Second, HPC centres frequently restrict outbound connectivity and privileged operations, which requires careful packaging of credentials, conditions data and ancillary services, or controlled egress via gateways/VPNs [25,26]. Third, filesystem characteristics (e.g., metadata performance of shared parallel filesystems) can affect containerised workloads; works report benefits from pre-warming caches and reducing small-file traversals via CVMFS subsetting [21]. Overall, prior art shows that virtualisation and containerisation are enabling technologies for HEP at scale, but that operational details, such as image design, cache topology and site policy integration, determine success.

2.4 Monitoring Frameworks for Distributed and HPC Systems

Operating a worldwide, heterogeneous computing fabric requires monitoring stacks that cope with high-rate telemetry, long retention for accounting and interactive troubleshooting. Over the last decade, the WLCG and the experiments have converged on open-source, stream-oriented architectures (search/index stores plus time-series plus dashboards) while retiring bespoke point solutions.

A first consolidation step was CERN's Unified Monitoring Architecture (UMA), which merged IT data-centre and grid monitoring into a common data pipeline and tooling set. UMA describes a layered design, from collection (agents and message buses) to transport and storage (e.g. search and analytics stores) to visualisation (dashboards and notebooks), explicitly sized for petabyte-scale monitoring and multi-tenant use by service teams and experiments [28]. Building on that, the MONIT programme documented the production deployment for WLCG: standardised ingestion, streaming, storage and alerting and shared dashboards for job throughput, data transfers and site/service health [29]. This shift enabled experiments and sites to de-duplicate tooling and rely on shared, operated services.

For grid-level, experiment-facing views, ATLAS developed PanDA's Monitoring Extension (BigPanDA), a web application that aggregates PanDA state at multiple abstraction levels (task, job, site, dataset) and exposes interactive exploration, historic trends and alarms used in daily operations [30,31]. BigPanDA's architecture and usage statistics (tens of thousands of requests per day at Run-2 scale) illustrate the need for responsive, indexed backends and caching in face of hundreds of millions of job records. Complementary ATLAS dashboards evolved for interactive visual analytics of production activity and site behaviour [32].

CMS has similarly reported a full monitoring stack for its distributed computing, partly based on CERN MONIT. The stack integrates streaming ingestion, a mix of time-series and document stores and dashboards/alerts for workload, data management and transfers; it also emphasises cross-correlation (e.g. mapping job failures to site incidents) and programmatic access for automated remediation [33,34]. Earlier CMS work on web-based monitoring already highlighted requirements for availability, drill-down and integrated authentication at LHC scale [35].

At the WLCG coordination level, the "Dashboards with Unified Monitoring" effort documented the migration of legacy experiment-specific tools into common services and dashboards tailored for operators and experiment shifters (transfer health, site availability, capacity and accounting). This work also discusses the selection of open-source components and the governance/operational aspects of running shared monitoring for a federated infrastructure [36].

Across these efforts, several patterns recur: centralisation of logs/metrics for correlation across subsystems; near-real-time visualisation for operations with long-term archival for accounting and planning; multi-tenant dashboards with role-based access; and Application Programming

Interface (API) that enables automated reaction (e.g. alarms to ticketing/blacklisting). For HPC integration, these stacks matter because they allow experiments to observe containerised payloads running under local schedulers, diagnose performance pathologies related to shared filesystems and networking and demonstrate non-interference with co-tenants.

2.5 Scheduling Strategies and Opportunistic Computing

Large distributed HEP workflows are typically dominated by many loosely coupled jobs. Two families of schedulers therefore meet in practice: high-throughput (late-binding, pilot-based) systems on the grid side and batch schedulers with queueing/fair-share on HPC systems. Efficient and non-disruptive use of resources in hybrid settings hinges on scheduling strategies that reconcile these models and on mechanisms that safely exploit slack capacity.

On the grid/HTCondor side, pilot systems implement late binding and opportunistic use of resources: placeholder jobs start on available slots and then pull real payloads. This design lets workload managers react to heterogeneous site conditions and failures and to prioritise production campaigns dynamically. In ATLAS, PanDA pilots are increasingly provisioned via Harvester, which interfaces pilots to external batch systems and enables elasticity across sites, including HPC backends with constrained policies [15]. CMS reports similar operational patterns when using network-segregated or policy-restricted resources, combining glide-in provisioning with targeted data staging and policy-aware submission to avoid interference with co-tenants [25,26].

On the HPC side, classic queueing strategies seek to maintain high utilisation while preserving fairness. Backfilling - running smaller jobs ahead of a large queued job provided its reservation start time is not delayed - remains foundational. Feitelson and Mu'alem Weil's analysis on IBM SP2 showed how backfilling improves utilisation and "predictability" (bounded waiting time variance) under realistic user run time estimates [37]. Later work refined the role of estimates and scheduler predictions, but backfilling persists as the key lever for throughput on shared clusters. Production batch systems such as Slurm implement these ideas alongside priority and fair-share policies [38].

Hybrid grid-HPC operation introduces lifecycle constraints uncommon on traditional grid sites. Jobs on HPC must conform to site wall time and maintenance cycles; when experiments encapsulate pilot+payload inside long-running allocations (e.g., containerised "drones"), operators often adopt draining - stopping acceptance of new payloads before allocation expires - to mitigate job loss. That practice protects primary workloads but leaves tail capacity idle. Several studies document strategies to reclaim such slack: adapting pilot provisioning to backfill windows discovered by the local scheduler [15]; shaping payloads (shorter tasks, finer granularity) to fit within remaining wall time, demonstrated in CMS workflows on restricted resources [25,26]; and dynamically integrating opportunistic resources by predicting resource usage to schedule in slack windows, as described in "Lightweight Dynamic Integration of Opportunistic Resources" [39].

Safe reclamation requires isolation. Here, cgroup-based policies integrated with container run times and HTCondor are critical: they cap CPU and memory for opportunistic jobs, enforce immediate yield/pre-emption and provide accurate accounting of resource usage. Measurements from production HTCondor+Singularity deployments show that cgroup enforcement achieves strong isolation with negligible overhead for typical scientific workloads, enabling aggressive opportunistic strategies without harming primary jobs [40].

2.6 Summary and Open Gaps

From the examined related work (Sections 2.1 through 2.5), the following key achievements stand out:

WLCG infrastructure and experiments (ATLAS, CMS, ALICE, LHCb) have successfully developed hybrid models integrating HPC, grid and opportunistic resources, with containerisation and monitoring now standard parts of the stack.

Tools like CVMFS, Singularity / Apptainer, Harvester and dashboards built on ELK / Grafana / time-series stores enable portability, reproducibility, observability and responsive operational control.

Scheduling techniques (late binding pilots, draining, backfilling, opportunistic pools, cgroup isolation) have been shown to reclaim idle CPU cycles while maintaining performance and respecting site policies.

Experiment-based case studies such as "Using ATLAS@Home to Exploit Extra CPU from Busy Grid Sites" [41] demonstrate that backfilling and volunteer/low-priority tasks can add significant utilisation (e.g. 15-42%) without harmful side-effects.

Tools to improve pilot-provisioning (such as LHCb's Distributed Infrastructure with Remote Agent Control (DIRAC) Site Director) show that performance of pilot-job systems remains an active area for throughput and latency improvements [42].

Yet, several open gaps persist; these are not fully addressed in the literature and align closely with the problem space of this thesis:

Lifecycle-aware scheduling at the edges of allocations Many studies mention draining and backfilling (e.g. ATLAS@Home) but less work has systematically quantified the idle CPU hours at the tail end of containerised worker allocations (such as on HPC resources under fixed wall time) and optimised policies to automatically schedule background jobs or pre-emptible work in those periods.

Monitoring & accountability for mixed environments While dashboards and monitoring for grid jobs are mature, less published work shows fine-grained observability of containerised

“drone” or pilot worker nodes within HPC allocations: visibility into metrics such as memory usage, I/O performance, container start-up delays, or tail idle time. Without this, optimising scheduling/policy decisions is harder.

Non-interference guarantees and resource isolation Although cgroup and container isolation are used, there is relatively little published evidence on guarantees (or even measurements) of non-interference when opportunistic/background jobs run alongside primary payloads, especially in HPC settings with shared filesystems, network, interconnect contention.

Policy and administrative constraints Access restrictions (network, software stack, privileged operations), site wall time limits, scheduling queue policies vary widely. The literature acknowledges these but often treats them as fixed constraints; few works propose adaptable scheduling or negotiation layers that can adapt to different site constraints dynamically.

Evaluation over HL-LHC-scale workloads Many studies are done with current Run-2 or early Run-3 workloads; as HL-LHC approaches, expected data volumes, simulation needs and resource heterogeneity will increase by an order of magnitude. Published work often lacks performance evaluation under those future scales, particularly in terms of efficiency of opportunistic use, waste at lifecycle tails, or monitoring overheads when scaling up.

Quantifying the trade-offs There is limited quantitative comparison between different approaches: e.g. trade-offs between container start-up time vs image size; between background job pre-emption cost vs idle time recovered; between monitoring granularity vs overhead. More studies are needed to guide scheduling policies based on real data.

Chapter 3

Previous Work

3.1 Introduction

The Master of Science (MSc) project that preceded this thesis focused on the characterisation of job execution at the WLCG Tier-2 site GoeGrid, operated in Göttingen and the development of a prototype monitoring infrastructure. The work aimed to understand how resource consumption patterns, particularly memory and network bandwidth, vary with the number of cores and job mix and to establish methods for predicting usage. This was motivated by the need to ensure efficient operation of grid resources and to prepare for the transition towards HPC-based infrastructures such as EMMY.

The project used one year of job execution logs from GoeGrid, analysing job resource usage across thousands of workflows. Statistical modelling was employed to identify trends in memory and bandwidth consumption, with regression techniques used to derive predictive functions for resource estimation. In parallel, a proof-of-concept monitoring stack based on the ELK framework was deployed to collect, index and visualise job execution data, providing operators with an integrated view of site performance.

This chapter summarises the main contributions of that work. Section 3.2 reviews the methodology and results of the resource consumption analysis, highlighting predictive models and their limitations. Section 3.3 outlines the design and deployment of the monitoring infrastructure, as well as the operational lessons learned. Together, these elements provide the foundation for the current thesis, which extends the analysis and monitoring beyond GoeGrid into the hybrid grid-HPC context of the NHR cluster EMMY.

References to related methodologies such as statistical regression [43], workload classification [44] and distributed monitoring frameworks [45] are included to situate the MSc contributions in a broader context.

3.2 Resource Consumption Analysis

The first component of the MSc project was a systematic analysis of one year of job execution data from the Göttingen Tier-2 site GoeGrid. The dataset covered thousands of jobs submitted by ATLAS through the PanDA system, including both production and user analysis tasks. Each record included metadata such as job duration, core count and memory and network usage. The primary goal was to identify patterns that could be used to predict resource demands based on job parameters, thereby supporting scheduling decisions and site planning.

To achieve this, the analysis employed regression-based techniques, inspired by methods such as LOESS smoothing for non-linear trend detection [43]. Job samples were binned according to the number of allocated cores and average memory and bandwidth consumption were computed within each bin. Regression functions were then fitted to model scaling behaviour. This approach revealed that memory usage typically increases sub-linearly with the number of cores, reflecting shared memory regions within multi-core tasks, whereas bandwidth demand often exhibits near-linear scaling with core count.

Figure 3.1 (adapted from the MSc report) illustrates these relationships. The plots show regression fits, highlighting the predictive potential of relatively simple models. However, variability within bins was significant, especially for user analysis jobs, which often display heterogeneous resource profiles.

In addition to regression, classification-inspired approaches were explored, following strategies similar to workload classification challenges in HPC [44]. Jobs were grouped into categories based on execution patterns, which improved prediction accuracy for resource estimation when compared to a global fit. This suggested that hybrid approaches, combining regression and classification, may be more effective for operational use.

The MSc project also emphasised the importance of understanding user behaviour in HPC and HTCondor contexts [46]. For instance, short exploratory jobs and long production runs show different scaling properties and sensitivities to system bottlenecks. By quantifying these distinctions, the project provided practical input for scheduling strategies such as backfilling [47] and opportunistic job execution.

The resource consumption analysis demonstrated that predictive models based on historical data can inform both local site optimisation and broader integration with HPC resources. At the same time, the observed variability highlighted the need for continuous data collection and adaptive modelling, which motivated the development of the monitoring framework described in Section 3.3.



Figure 3.1: Job share every month from 01.08.2023 to 31.07.2024 (top). The corresponding calculated bandwidth (middle). And the corresponding calculated memory (bottom). Calculated for 20000 cores. Processing type describes the type of the submitted job.

3.3 Monitoring Infrastructure Setup

The second component of the MSc project focused on setting up a monitoring framework capable of supporting both retrospective analysis and near real-time observability of job execution on GoeGrid. While the resource consumption study in Section 3.2 relied on batch-collected data, long-term sustainability of such efforts required automated pipelines and dashboards aligned with best practices in distributed systems monitoring.

The monitoring setup was based on the deployment of the ELK stack, chosen for its scalability and ecosystem maturity. Logstash agents collected job metadata, scheduler logs and system-level statistics from worker nodes, which were forwarded to Elasticsearch for indexing. Kibana dashboards provided interactive visualisation of resource usage patterns, complementing the static regression plots produced during the resource analysis phase. This design enabled operators to quickly identify anomalies such as misconfigured jobs, unbalanced resource allocation or filesystem/network contention.

A key aspect was aligning the monitoring with established principles from large-scale distributed system operations. Google’s Site Reliability Engineering (SRE) guidelines emphasise the “four golden signals” (latency, traffic, errors, saturation) as minimal metrics to ensure service reliability [45]. These ideas were adapted to the scientific computing context, where analogous metrics included job queue latency, throughput of submitted jobs, failure rates and utilisation of cores, memory and I/O.

In addition, the MSc project experimented with integration of external knowledge sources, such as cloud storage logs (e.g. Amazon Web Services, Inc. (AWS) S3 [48]) and interoperability frameworks like Reva [49], which are increasingly relevant as HEP workflows span multiple resource providers. While only exploratory at this stage, these experiments indicated the feasibility of federating site-level monitoring with wider data ecosystems such as ScienceMesh [50].

The monitoring pipeline also incorporated lessons from research on user behaviour and workload diversity [46], which showed that variability in usage patterns requires flexible dashboards and drill-down capabilities. For instance, differentiating between production and user analysis jobs allowed the system to highlight distinct failure modes and resource demands. Furthermore, security considerations in distributed logging [51] informed the system’s design, ensuring sensitive logs were collected with appropriate filtering and access control.

Taken together, the MSc monitoring infrastructure provided a functional prototype of an observability stack for mixed HTCondor - Slurm sites. Although deployed at a local Tier-2, the design principles of log centralisation, metric-based alarms and user-aware visualisation are transferable to hybrid environments such as those targeted in this thesis.

Chapter 4

Design and Methodology

4.1 Introduction

The preceding chapters reviewed the state of the art in distributed and HPC computing for HEP, as well as the analysis and monitoring work that formed the basis of this project. Building on these foundations, this chapter presents the design choices, implementation details and methodological considerations of the work conducted in this thesis.

The guiding principle has been to adapt existing practices in large-scale scientific computing to the specific requirements of the Göttingen environment, while ensuring that the solutions remain interoperable with the broader WLCG ecosystem. Three main areas of development were pursued. First, a monitoring pipeline was established to integrate heterogeneous sources of system and application metrics into a consolidated framework, enabling both operational insight and long-term performance analysis. Second, methods for running background jobs were designed to exploit idle resources opportunistically, while respecting HPC scheduling constraints and minimising interference with primary workloads. Finally, parts of the existing orchestration codebase were refactored to improve modularity, maintainability and adaptability to the hybrid grid-HPC use case.

The following sections describe these contributions in detail. Section 4.2 outlines the design and deployment of the monitoring stack. Section 4.3 discusses the approach to opportunistic scheduling of background jobs. Section 4.4 summarises the refactoring of the orchestration codebase. Together, these developments form the methodological backbone of the work presented in this thesis.

4.2 Monitoring Pipelines

The monitoring pipeline was designed to provide continuous, structured and scalable visibility into the execution of HTCondor workloads on the NHR cluster. Its purpose is twofold: to expose

detailed system- and job-level metrics for analysis and to integrate seamlessly with modern log aggregation and visualisation stacks, in particular the ELK stack (discussed in Section 3.3 and deployed during the MSc project period). The architecture follows a layered approach, starting with lightweight node-local agents, followed by log shipping and centralised parsing and, finally, culminating in storage and visualisation. This section details the reasoning, design decisions and implementation of each component in the pipeline.

4.2.1 Architecture and Design Rationale

The system adopts a hybrid deployment model that places agents either on worker nodes or on the monitoring node, depending on where authoritative data is located. Figure 4.1 illustrates the overall communication topology. On worker nodes, network traffic counters are inherently local and must be collected in situ. Conversely, Condor slot states, scheduling decisions and lifecycle events are already aggregated at the scheduler and therefore can be queried centrally. Similarly, power consumption data can be obtained from external Intelligent Platform Management Interface (IPMI) managed by the cluster vendor, rather than through node-resident daemons. This separation minimises overhead on worker nodes while ensuring that monitoring remains complete and consistent.

Data transport is deliberately file-based in the first stage: agents generate newline-delimited JavaScript Object Notation (JSON) records written to local log files. This choice avoids the complexity of direct streaming and provides resilience against transient failures of downstream collectors. Filebeat, a lightweight shipper, forwards logs to Logstash, which parses and enriches them before indexing in Elasticsearch. Kibana dashboards on the monitoring node provide interactive access to the metrics. Figure 4.2 gives a linear view of this flow for the network reader component.

4.2.2 Condor Reader

The Condor reader is a C++ daemon deployed centrally on the monitoring node. It periodically queries the HTCondor scheduler for slot and machine information using the `condor_status` tool. The implementation avoids per-node agents, since all required information about job slots, host activity and scheduling policies is already aggregated at the scheduler. This design reduces deployment complexity and eliminates redundant monitoring load on worker nodes.

The reader produces three classes of logs:

- Configuration logs: capture changes in slot flags and static configuration attributes, useful for detecting policy or accounting changes,
- Status logs: record slot lifecycle changes, including transitions between states (`Owner`, `Unclaimed`, `Claimed`) and activities (`Idle`, `Busy`, `Drained`),
- Metrics logs: provide periodic counters, including available and total CPUs, disk space, child

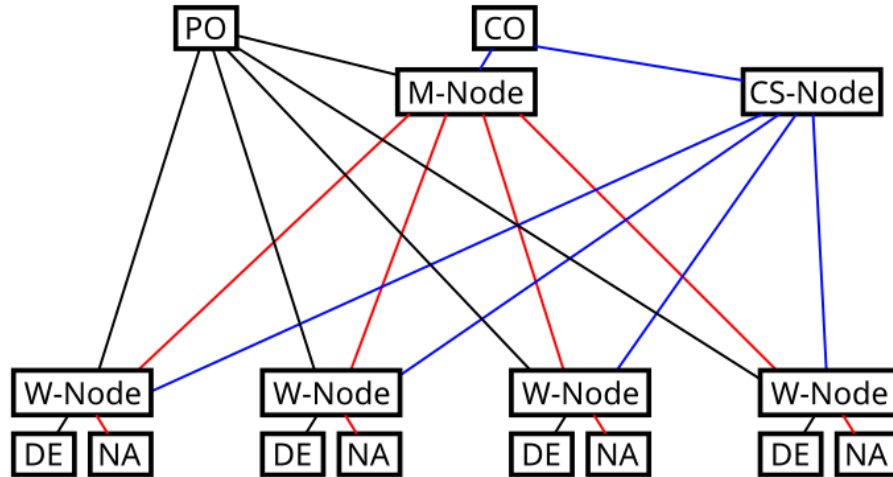


Figure 4.1: Component-level communication: Network Agents (NA) run on Worker Nodes (W-Node), Condor reader/observer (CO) and Power reader/observer (PO) run centrally on the Monitoring Node (M-Node). PO collects data from Dell Power/Energy reporting of each W-Node

process accounting groups and activity distributions.

A notable feature is lifecycle caching: hosts are identified by their name and daemon start time. Their last known phase (*alive*, *draining*, *dead*) is tracked across monitoring intervals. This enables accurate detection of draining slots and nodes removed from the pool. In addition, the reader supports user-defined flag specifications to extract boolean or numeric Condor attributes (e.g., site-specific start expressions).

Power metrics are integrated through pattern-based mapping between Condor hostnames and IPMI endpoints. For each matching rule, the reader issues an `ipmi` query to obtain maximum observed power draw during the sampling period. This bridges scheduler-level slot data with physical consumption measurements provided by the vendor.

All output is formatted as structured JSON lines with timestamps, ensuring compatibility with downstream log processing. Example fields include `cpus_free`, `children_busy`, `group_cpus@atlas` and `power_max_watts`. The design is inspired by earlier efforts to provide detailed Condor monitoring for large infrastructures [46,52] but emphasises log-stream integration and lifecycle tracking.

4.2.3 Network Reader

While the Condor reader centralises scheduler state, the network reader provides decentralised collection of interface-level traffic statistics. Implemented in C++, it reads the host network device counters on each worker node at configurable intervals (60 seconds by default), calculates deltas

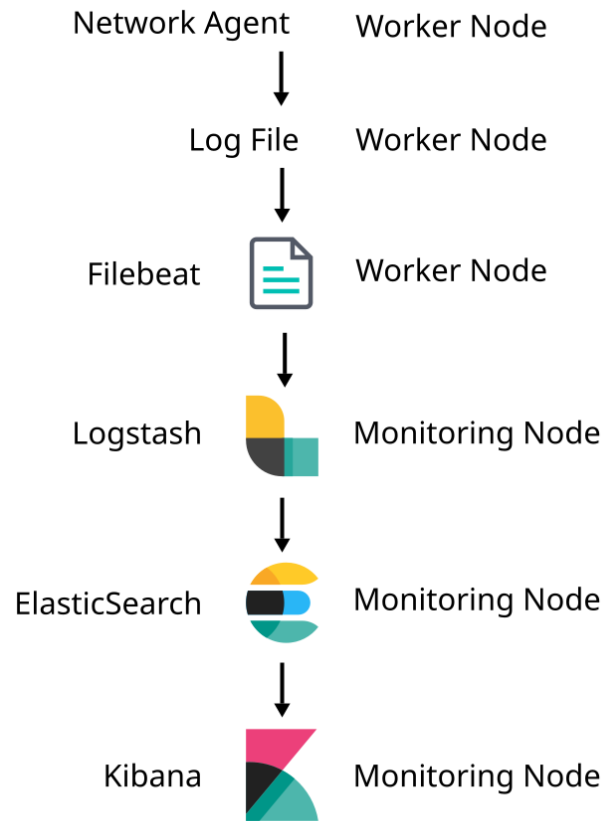


Figure 4.2: Example of the network agent data pipeline, showing log file generation, Filebeat shipping, Logstash parsing, Elasticsearch indexing and Kibana visualisation. On the right the deployment location of each component is described.

using wraparound-safe subtraction and writes results to structured logs.

Key features include:

- **Interface discovery:** if no interfaces are specified, all available interfaces are discovered from `/proc/net/dev`,
- **Pattern-to-tag mapping:** an external configuration file maps Internet Protocol (IP) address patterns to semantic tags (e.g., Wide Area Network (WAN), InfiniBand, management, internal, external, etc.). Interfaces are automatically tagged based on their addresses, allowing traffic to be grouped by function,
- **Synthetic counters:** users may define expressions that combine counters across tags or interfaces (e.g., `!WAN = eth0 + eth1`), evaluated at runtime. This provides flexibility to monitor aggregate traffic without modifying the agent code,

- The configuration file is wildcard (*) friendly. Users can specify IP address `123.123.*` and all IP addresses matching the pattern will be selected and then tagged with the provided tag (e.g. `external`). On practice if there are 10 nodes and their external addresses are `123.123.0.[1-10]`, the agent on each of the nodes will select only one interface. This makes it easy to have just one configuration file for the cluster instead of writing one line different configuration files for each node,
- Tagging is implemented in order to enable data aggregation even in case of different names of interfaces serving the same purpose (e.g. `eth0` on node A and `eth1` on Node B both are for the external traffic),
- Selective logging: Patterns defined with `!` (e.g. `!WAN`) are synthetic entries composed of other entries (e.g. `eth0` and `eth1`) and are saved. Tags defined with a `!` prefix (e.g. `!eth0`) are treated as components of synthetic entries and excluded from raw per-interface logs. They can be kept by removing `!` symbol. Example entries in config can look like:

```
# pattern          tag
10.123.*           !wireless_001
10.124.*           !wireless_002
!wireless_001+wireless_002  wireless
```

The agent outputs JSON lines containing timestamps, interface names, byte deltas and optional tags. These are collected via Filebeat and indexed into Elasticsearch, enabling dashboards that track both raw and aggregated bandwidth consumption. This functionality builds on long-standing experience with distributed network monitoring in HEP sites [29,35], but provides a configurable, container-friendly implementation tailored for hybrid grid-HPC environments.

4.2.4 Integration with the ELK Stack

Both readers follow the same downstream path: logs are collected by Filebeat, parsed by Logstash, indexed into Elasticsearch and visualised with Kibana. The choice of ELK is consistent with the practice in WLCG and experiment operations, where central monitoring infrastructures have converged on open-source pipelines with search and analytics back-ends [28,29,36]. By emitting structured JSON at the source, the readers minimise parsing complexity and support schema evolution with backward compatibility.

A critical design aspect is the decoupling of metric collection from transport. Agents remain functional even if the monitoring node or Elasticsearch cluster is temporarily unavailable, since logs are first persisted locally. This ensures resilience and simplifies debugging, as raw logs can be inspected before ingestion. Additionally, using Filebeat as the shipping layer avoids embedding network transport logic in the agents themselves, keeping them small, auditable and easy to maintain.

4.2.5 Summary

The monitoring pipelines provide a flexible and resilient mechanism to observe workload execution at both the scheduler and network levels. Centralised Condor readers capture scheduler state, lifecycle phases and power metrics without requiring per-node deployment. Distributed network readers observe local traffic counters and support aggregation through tagging and expressions. Both pipelines integrate with the ELK stack through JSON log emission, ensuring compatibility with existing monitoring infrastructure in HEP computing. The design emphasises minimal intrusion on worker nodes, configurability through external files and robustness under production conditions, thereby laying the foundation for detailed performance analysis in later chapters.

4.3 Background Jobs

The background job framework was designed to exploit otherwise idle cycles on EMMY nodes without interfering with the execution of foreground HPC workloads. This approach leverages the concept of virtual worker nodes (drones), which are Slurm-launched containers configured to register into the GoeGrid HTCondor pool as standard worker nodes. Within these drones, background tasks are implemented as a dedicated slot governed by strict cgroup controls and PanDA queue separation. The goal is to enable opportunistic harvesting of idle capacity, especially during draining phases, while ensuring fairness and isolation for primary jobs.

4.3.1 Architecture and Drone Deployment

Drones are launched on EMMY through Slurm allocations that launch containers with HTCondor `Start Daemon (startd)` processes. The launch script installs and configures `cvmfsexec`, required because CVMFS must be mounted on the host. Using `apptainer`, the container is started with explicit bind mounts for `/cvmfs`, Condor configuration files, temporary directories and log paths. Inside, only the `MASTER` and `STARTD` daemons are run, with `tini` as `init` to ensure proper signal handling and process reaping.

Each container registers back to the GoeGrid Condor central manager and appears indistinguishable from a physical worker node (except for human readable hostname pattern difference), with names such as `slot1@c-ssdlocal-xxxx.local`. From the scheduler's perspective, these drones extend the pool dynamically using HPC resources, while respecting site-level constraints.

4.3.2 Slot Layout and Queue Separation

Within each drone, two partitionable slots are defined:

- **slot1**: reserved for foreground jobs, attached to the PanDA EMMY queue - production jobs,

- **slot2**: reserved for background jobs, attached to the PanDA EMMY_BKG queue - lightweight background jobs.

Queue separation is enforced through Condor start expressions and PanDA flags. Jobs are submitted with attributes `AllowHPC` or `AllowBKGHPC`, while slots are labelled with `IsHPC` and `IsBKGHPC`.

The start policy below gates when a slot may start a job and which jobs it may accept:

```
START = (StartJobs == True) &&
        ((SlotID == 1 && StartSlot1 && (TARGET.AllowHPC)) ||
         (SlotID == 2 && StartSlot2 && (TARGET.AllowBKGHPC)))
```

Key elements:

- **START**: The HTCondor startd policy that must evaluate to `True` for a claim to be made and for a job to start on the slot. If `START` is `False`, the slot remains `Unclaimed/Idle` or transitions to draining depending on other settings,
- **StartJobs**: A site-controlled boolean “master switch”. The expression `(StartJobs == True)` uses the Classified Advertisement (ClassAd) “is” operator `==` so it is only `True` when `StartJobs` exists and is `True`; if `StartJobs` is undefined or `False`, the whole `START` becomes `False`. This allows pausing all starts without editing the full policy,
- **SlotID**: The numeric identifier of the executing slot (e.g., 1 for the partitionable foreground slot; 2 for the background slot). It lets a single policy branch differently per slot,
- **StartSlot1, StartSlot2**: Per-slot toggles (booleans) that can be changed at runtime (e.g., via a tiny config drop-in and `Signal Hang Up (SIGHUP)`). In the experiments, these flips implement the lifecycle: run foreground first (`StartSlot1=True, StartSlot2=False`), then switch to background to fill the drain (`StartSlot1=False, StartSlot2=True`) and, finally, disable both (`StartSlot1=False, StartSlot2=False`),
- **TARGET. . . .**: Attributes from the jobs’ ClassAd. Here, jobs carry flags set by the ARC-CE depending on the PanDA queue identification:
 - `TARGET.AllowHPC`: set for jobs from the EMMY (foreground) queue.
 - `TARGET.AllowBKGHPC`: set for jobs from the EMMY_BKG (background) queue.

Slots are labelled with `IsHPC/IsBKGHPC` (advertised machine attributes) and the site policy uses the `TARGET.` flags to accept only the intended stream,

- **Logic flow**: The outer AND requires the global switch `StartJobs` to be `True`. The inner OR selects:
 1. Foreground execution: `SlotID==1 && StartSlot1 && TARGET.AllowHPC`.

2. Background execution: `SlotID==2 && StartSlot2 && TARGET.AllowBKGHPC`.

Thus, foreground jobs can only start on slot 1 while its toggle is on and background jobs only on slot 2 while its toggle is on.

Operationally, this enforces queue separation between native GoeGrid, EMMY (HPC) and EMMY_BKG (background) streams and it enables the controlled “drain-fill” phases used in the experiments.

4.3.3 cgroup-Based CPU Management

Foreground and background slots are isolated through cgroup v2 controls. The new Condor configuration specifies:

```
BASE_CGROUP = ../condor.service
CGROUP_MEMORY_LIMIT_POLICY = soft
CGROUP_ALL_DAEMONS = True
STARTER_TRACK_GROUPS = True
CREATE_CGROUP_WITHOUT_ROOT = True
```

- **BASE_CGROUP = ../condor.service:** anchors the Condor-managed hierarchy under the `condor.service` slice inside the enclosing Slurm job cgroup. This ensures that all Condor slots are created as subgroups of the parent Slurm allocation, keeping accounting and limits consistent with the batch system,
- **CGROUP_MEMORY_LIMIT_POLICY = soft:** instructs Condor to treat memory limits as soft constraints. Jobs are monitored against their requested memory, but Condor does not enforce hard termination when the limit is exceeded. This is essential for background jobs, which opportunistically use resources but should not crash the container or node if temporary fluctuations occur,
- **CGROUP_ALL_DAEMONS = True:** places not only job payloads but also Condor daemons (`startd`, `shadow`, etc.) into the Condor cgroup tree. This provides consistent resource accounting and prevents daemons from escaping the Slurm allocation boundaries,
- **STARTER_TRACK_GROUPS = True:** enables Condor starters to explicitly track and update their jobs’ cgroup membership. Without this, processes spawned by payloads may escape tracking. With it, every child of a job is accounted for under the correct slot subgroup, which is crucial for accurate CPU usage measurements,
- **CREATE_CGROUP_WITHOUT_ROOT = True:** allows Condor to create and manage cgroups without requiring root privileges inside the container. This is critical because drones run as unprivileged users in the Slurm job context. The parameter ensures proper subgroup creation even when Condor cannot escalate privileges.

The `condor.service` subtree is created inside the Slurm job's systemd slice:

```
/sys/fs/cgroup/system.slice/slurmstepd.scope/job_<slurm_job_id>/condor.service
```

Within this subtree, Condor dynamically spawns per-slot/job cgroups, such as:

```
condor.service/condor_tmp_local_condor_execute_slot1_[x-xxx]@c-ssdlocal-xxxx.local
```

Here, x is any digit between 0 and 9. Each cgroup is assigned a `cpu.weight` proportional to the number of CPUs requested: $N_{\text{cpus}} \times 100$. Thus, an eight-core job receives a weight of 800, while a single-core job receives 100. Foreground jobs are naturally scheduled with their full requested weight, ensuring proportional access to busy CPUs. Background jobs, however, are submitted as single-core tasks (weight 100) but configured to opportunistically consume additional CPUs if idle. The cgroup scheduler enforces that they yield immediately under contention, thereby allowing them to backfill only genuine slack capacity.

4.3.4 Lifecycle Control and Experiment Modes

To study the effectiveness of background filling, multiple operational modes were explored:

- **Control (no background):** slot2 disabled, only slot1 processes foreground jobs.
- **Background always-on:** slot2 active from the start, continuously filling idle cores.
- **Background during drain:** slot1 is deactivated (drained), while slot2 continues running, filling CPUs left idle by draining.
- **Background on-demand:** slot2 is activated only after draining begins, targeting the filling of released capacity.

Slot activation is controlled by toggling the variables `StartSlot1` and `StartSlot2` in the Condor configuration file `99dynamic.conf`, followed by a `SIGHUP` signal to `condor_startd`. This provides a reproducible mechanism to experiment with background job behaviour under different lifecycle conditions. Timeline plots of slot activations and job placements (included in Chapter 5) illustrate the CPU occupancy patterns across these modes.

4.3.5 Validation

To validate that background jobs filled idle cores without disrupting foreground tasks, a dedicated tool, `condor_cpu_watch`, was developed. It traverses the `condor.service` cgroup tree, identifies slot-specific job subgroups and periodically reads `cpu.stat`. This provides time-resolved accounting of CPU usage per slot, independent of Condor's own counters.

This validation ensured that:

1. background jobs indeed expanded beyond one CPU only when free cores were available,

2. foreground jobs maintained their requested CPU shares regardless of background activity,
3. draining phases showed the intended effect of background jobs opportunistically filling released CPUs.

These measurements were put into the context of job mix on the drones at the time, in order to verify that any difference in CPU efficiency was not caused by difference in job nature.

4.3.6 Summary

Background jobs are an additional slot attached to a distinct PanDA queue and controlled via cgroup policy. By combining Condor start expressions, PanDA queue flags and CPU weight enforcement, the design achieves strict separation and fair sharing. Background jobs are thereby confined to opportunistic capacity: they fill otherwise idle CPUs during drains or low utilisation, but immediately yield under contention. Validation with `condor_cpu_watch` confirmed the effectiveness of this strategy. Together, this system provides a controlled method for integrating background workloads into hybrid grid-HPC deployments without compromising primary job execution.

4.4 Codebase Refactoring

The refactoring of the drone management codebase aimed to transform a complicated set of ad hoc scripts into a modular, maintainable and declarative system. The functional goal remained unchanged: deploy containerised HTCondor worker nodes (drones) on the NHR cluster such that they register in the central GoeGrid HTCondor pool. The new design emphasises reproducibility, portability and safety. This section documents the methodology and control mechanisms of the refactored system, serving as both a technical description and a practical guide for configuration and operation.

4.4.1 Methodology and Design Principles

The original implementation consisted of several shell and Python scripts orchestrated in sequence: prepare directories, install CVMFS, build an Apptainer container and, finally, start HTCondor. While functional, this model suffered from duplicated logic, hardcoded paths and site-specific edits. The refactoring was guided by three principles:

- **Declarativity:** operations such as copying or creating directories are expressed in configuration, not in code,
- **Idempotency:** the workflow can be re-run safely without manual cleanup,
- **Portability:** site-specific information is isolated into configuration files, enabling re-use across environments.

The refactored architecture is organised into a single Command-Line Interface (CLI) tool (`drone.py`) and helper modules under `common/helpers/`, separating concerns such as configuration parsing, container orchestration and cgroup setup.

4.4.2 Control via Command-Line Interface

The entry point `drone.py` exposes two subcommands:

- `install` Prepares the working directory and builds the container image. Typical use when creating a new drone installation.
- `submit` Deploys the prepared environment, installs CVMFS with `cvmfsexec`, sets up bindings and cgroups and launches the container with HTCondor. This is the command used to actually run a drone.

Both commands accept:

- `-config <path>` Path to the YAML Ain't Markup Language (YAML) configuration file. Defaults to `drone_configs/default.yml`.

The `submit` command additionally accepts:

- `-lifetime <seconds>` Lifetime of the drone in seconds. Overrides the value in the configuration. The lifetime is passed to the start script inside the container and controls when the drone shuts down.

These options define the high-level control surface for users: prepare an environment with `install`, then run it with `submit`.

4.4.3 Configuration File Reference

The YAML configuration file defines all aspects of deployment. Every field is interpreted at runtime, with placeholders (e.g. `<subdirectories.htcondor>`) expanded dynamically. Below is a detailed explanation.

General fields.

- `node_label`: short label describing the node type, used in naming conventions.
- `server_label`: descriptive name for the server configuration, e.g. `NHRGoe_standard96`.
- `home_path`, `main_path`, `install_path`: paths controlling where code lives, where it is built and where installations are deployed. Placeholders like `<paths.main>` are resolved automatically.
- `install_basename`: base directory name used when generating new working directories.

- `env_local_tmpdir`: environment variable that may override the default installation path when deploying.
- `hostname`: hostname to assign to the drone container. If empty, one is generated automatically.
- `lifetime`: default drone lifetime in seconds (overridable by CLI argument).

Subdirectories. The `subdirectories` block defines relative paths for different functional groups:

- `container, def, sif`: directories for container definitions and images,
- `cvmfs, cvmfsexec`: CVMFS configuration and executables,
- `htcondor`: configuration for HTCondor,
- `shared`: shared files (scripts, hosts, configs),
- `grid_userdata`: account and passwd/group data.

File operations. Declarative lists describe what to copy, remove, create and `chmod`:

- `copy_list_build`: files or directories copied at build time (e.g. container def files),
- `copy_list_deploy`: files copied at deploy time (e.g. HTCondor configs, CVMFS configs, shared scripts),
- `rmdir_list`: directories to delete before redeployment (e.g. logs, lock, run),
- `mkdir_list`: directories to create,
- `chmod_list`: mode assignments for directories (e.g. 777 on logs, lock, tmp).

CVMFS repositories.

- `repos`: list of CVMFS repositories to mount through `cvmfsexec`, e.g. `atlas.cern.ch`, `sft.cern.ch`.

HTCondor block.

- `config`: name of the active HTCondor configuration (subdirectory under `htcondor/`),
- `cgroups_enabled`: enables integration with cgroup,
- `parent_cgroup`: template for the parent Slurm job cgroup path, e.g. `/system.slice/slurmstepd.scope/job_{slurm_job_id}`,
- `cgroup_env_gets`: list of environment variables to substitute into the template (e.g. `SLURM_JOB_ID`).

- `cgroup`: name of the HTCondor unit inside the parent cgroup, typically `condor.service`.
- `start_script`: path to the script invoked inside the container to start HTCondor.

Container block.

- `app`: container runtime, typically `apptainer`.
- `def_file`: Apptainer definition file used to build the container.
- `sif_file`: resulting Singularity Image Format (SIF) file.
- `replace_token`: placeholder replaced with `main_path` inside the `def` file.
- `bind`: list of bind mounts, mapping host paths to container paths. Defaults include CVMFS, HTCondor logs/run/lock directories, shared scripts and hosts file.
- `flags`: list of container flags. Flags can be added (e.g. `-writable-tmpfs`) or removed using prefixes (`omit:`, `remove:`, `!` e.g. `omit:-some-unwanted-flag`).

4.4.4 Cgroup Integration

The refactored system ensures that HTCondor runs inside the correct Slurm job cgroup. The helper `cgroups.py` expands the `parent_cgroup` template with runtime variables (such as `SLURM_JOB_ID`), creates a dedicated `condor.service` subgroup and exports the path into the container environment. Compared to the original shell implementation, which embedded fragile string concatenations, this approach ensures safety (refusing to operate outside `/sys/fs/cgroup`) and reproducibility.

4.4.5 Comparison with Original Implementation

Table 4.1 summarises the differences. The refactored system replaces hardcoded scripts with a clean separation of concerns and a declarative control surface.

Aspect	Original Implementation	Refactored Implementation
Entry point	Multiple shell and Python scripts	Single CLI with <code>install/submit</code>
Configuration	Hardcoded paths and variables	YAML with placeholders and runtime expansion
File operations	Inline shell commands	Declarative <code>copy/rmdir/chmod</code> lists
Cgroup setup	Path concatenation in scripts	Template expansion with environment-driven substitution
Container flags	Static command lines	Composable list with add/remove semantics
Portability	Site-specific edits required	Configurable across sites via YAML
Idempotency	Manual cleanup required	Automated cleanup and confined operations

Table 4.1: Comparison of original and refactored drone management implementations.

4.4.6 Summary

The refactoring turned a complex collection of scripts into a structured, declarative and portable codebase. Users interact only via two CLI commands and a YAML configuration. Every configuration field is explicitly defined, from directory layout to container flags and CVMFS repositories. The methodology emphasises idempotency, portability and safety, while providing a control mechanism that is both expressive and minimal. This foundation supports reproducible drone deployment and prepares the ground for the evaluation presented in the next chapter.

Chapter 5

Results

5.1 Monitoring Coverage

The monitoring stack provides a comprehensive view of the cluster and drone activity, combining data from multiple agents into a single ELK environment. The integration with Kibana enables interactive dashboards (Fig. 5.1) where administrators can drill down to individual worker nodes, correlate metrics and validate the correct behaviour of queue management.

At the network level, the monitoring agents deployed on the worker nodes report detailed traffic information. This enables inspection of both aggregate bandwidth patterns and per-node traffic, which is critical for identifying bottlenecks and ensuring that data-intensive HEP workflows do not overwhelm particular links.

In parallel, the power monitoring pipeline collects and aggregates metrics reported by Dell's hardware interfaces. This provides a per-node power consumption view, which is essential for validating the energy footprint of the deployment and for correlating workload intensity with electrical demand. By integrating these streams, the dashboards allow combined inspection of utilisation and consumption, providing an immediate indication of how effectively the hardware is being used.

The state of the worker nodes within the scheduler lifecycle is also directly visible. A dedicated dashboard shows which nodes are in Alive mode and which have transitioned to Draining. An example plot (Fig. 5.2) illustrates this separation, where draining nodes are clearly identified as they prepare to retire running jobs. This view is essential for validating that the scheduler interacts correctly with the underlying HPC batch system and that drones behave as expected when approaching their walltime limit.

In addition, the monitoring stack tracks the PanDA queue flags attached to each drone. These flags represent the various PanDA queues, which differ mainly by their data source. Since correct

flagging is required in order to track drones from different queue, with the purpose of accounting, visual confirmation within the monitoring environment is useful. A dedicated dashboard (Fig. 5.3) shows the active flags across drones. This plot confirms that queue control policies are enforced at runtime and that jobs are always matched to the appropriate resources.

Overall, the monitoring coverage demonstrates that the pipelines provide a complete picture of network usage, power draw, node lifecycle states and queue assignment. The combination of these dashboards establishes both operational observability and experimental validation of the methods developed in Chapter 4.

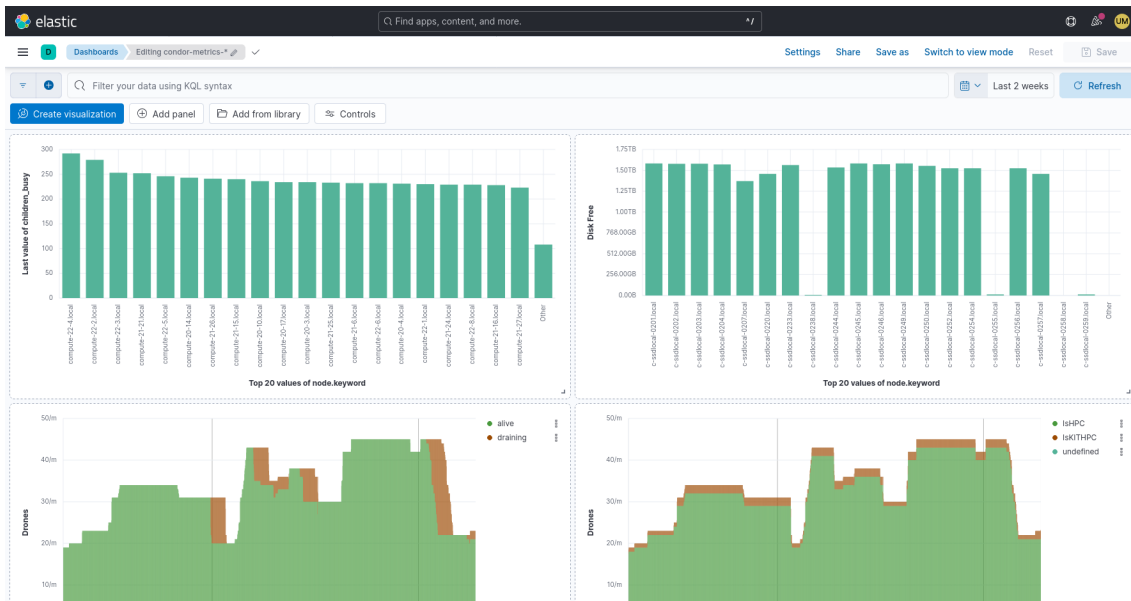


Figure 5.1: Overview of the Kibana monitoring interface, combining metrics from Condor readers, network readers and power monitoring into interactive dashboards.

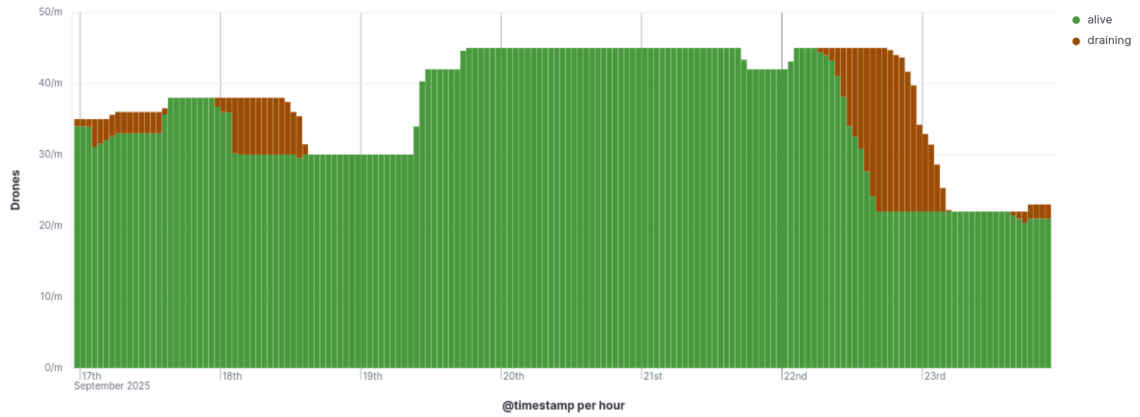


Figure 5.2: Lifecycle state dashboard, showing nodes in Alive and Draining modes as reported by the scheduler integration.

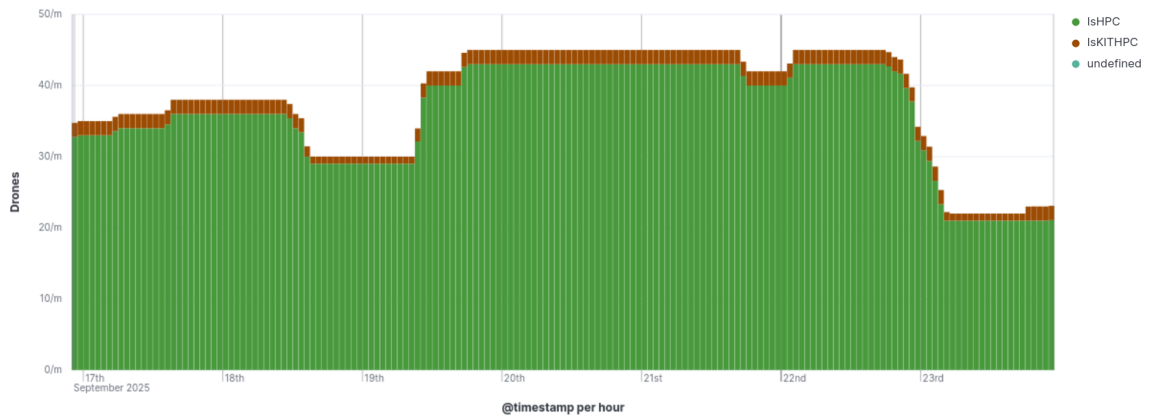


Figure 5.3: Queue flag dashboard, displaying PanDA queue attributes attached to drones and confirming correct separation of jobs across different data-source queues.

5.2 Impact of Background Jobs

To assess the impact of background jobs on foreground workloads, a series of controlled experiments was performed using the drone deployment on the NHR cluster. Each drone was configured with two slots: `slot1` for foreground jobs and `slot2` for background jobs. The experiments varied the activation of `slot2` to evaluate three distinct scenarios:

- Control group: only `slot1` active, no background jobs,
- Background slot active from launch: `slot2` enabled throughout job execution,
- Background slot active during draining: `slot2` enabled only when `slot1` began draining.

CPU utilisation was tracked using the `condor_cpu_watch` tool, which observes cgroup counters

of Condor slots. This provided a fine-grained view of how background jobs filled idle CPUs and whether they interfered with foreground jobs. Figure 5.4 shows that the job share across the initial experiments (Fig. 5.5 and 5.6) was consistent.

In the control group, CPU utilisation before draining was consistently close to 100% with no observable gaps (Fig. 5.5). When background jobs were enabled from the start, utilisation also remained close to full capacity (Fig. 5.6). However, interference was visible: because background jobs are scheduled as single-core tasks with a low cgroup weight, they competed with single-core foreground jobs for CPU time. This effect manifested as a temporary drop in CPU usage attributed to foreground jobs (0–20 CPUs), which then recovered once the background tasks completed. The efficiency plots confirmed this effect, showing an average 5% reduction in foreground CPU efficiency when background jobs were active (Fig. 5.7).

These observations motivated a third experiment, where background jobs were activated only during draining. In this configuration, background tasks successfully filled otherwise idle CPUs during the retirement phase. However, CPU utilisation before draining revealed occasional gaps even in the absence of background jobs (Fig. 5.8). This indicated that foreground workloads alone do not always saturate all available resources and that the initial full utilisation seen in the earlier experiments was an exception caused by the particular job mix at the time. To check this, the job mix of the third experiment was analysed as well (Fig. 5.9).

Repeating the three scenarios confirmed these trends (Fig. 5.10, 5.11, 5.12): foreground workloads alone sometimes leave idle CPUs, background jobs from launch consistently interfere with single-core foreground tasks and background jobs during draining avoid most interference while still recovering wasted capacity in the final phase of execution. The job mix (Fig. 5.13) and CPU efficiency (Fig. 5.14, 5.15) were also analysed. As with the previous attempt to launch background jobs from the start, there can be seen drop of CPU efficiency of $\sim 4\%$.

The experiments demonstrate that background jobs can improve aggregate utilisation of the HPC resources, but they must be carefully confined to draining phases to avoid reducing efficiency of primary production workloads.

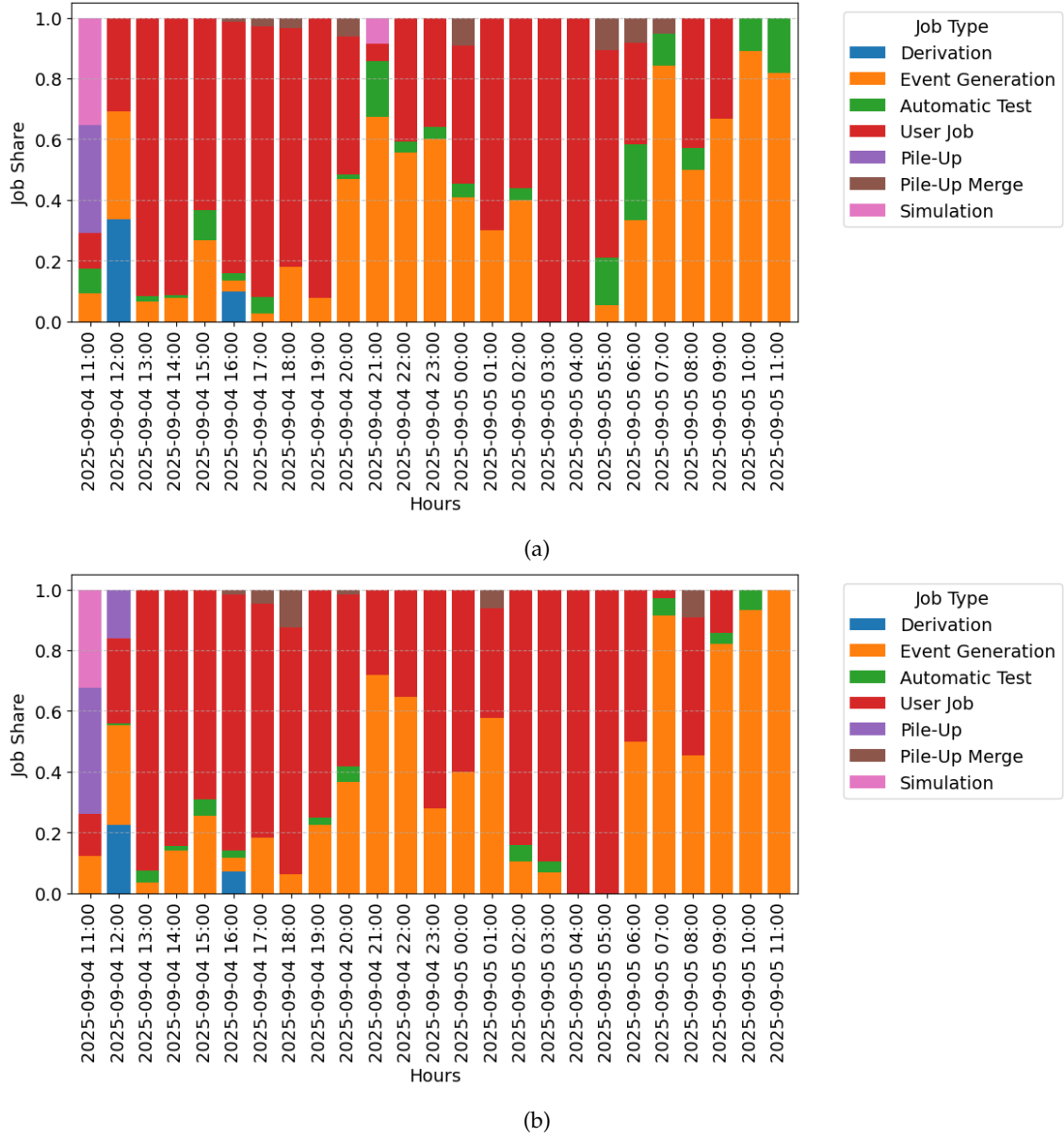


Figure 5.4: Job mix of foreground jobs with (a) and without (b) background jobs on the node. Background jobs launched from the start. It can be seen that during this period there was a lot of User Jobs, which are not typical production jobs and do not have standard resource consumption pattern.

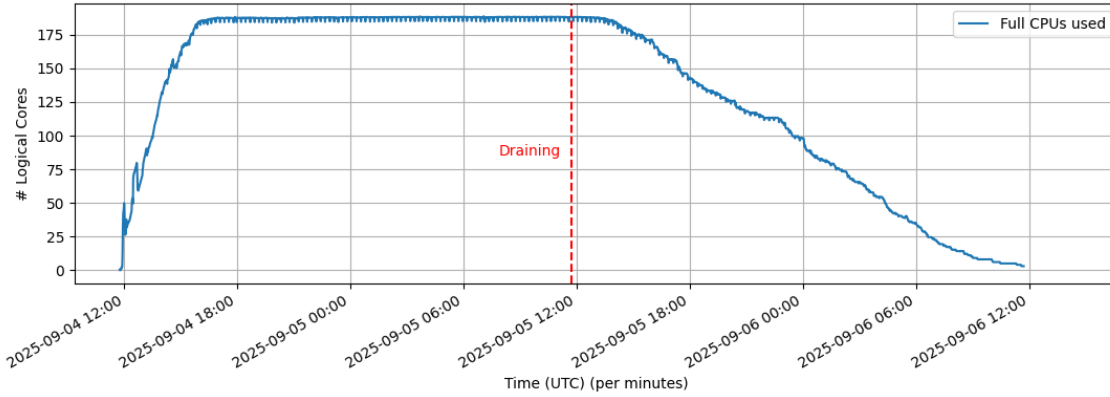


Figure 5.5: Full CPUs utilised on drones with no background jobs. Before draining the foreground jobs consume all available 192 CPUs. When the draining starts, CPU usage starts to gradually decline, due to jobs finishing and vacating the drone.

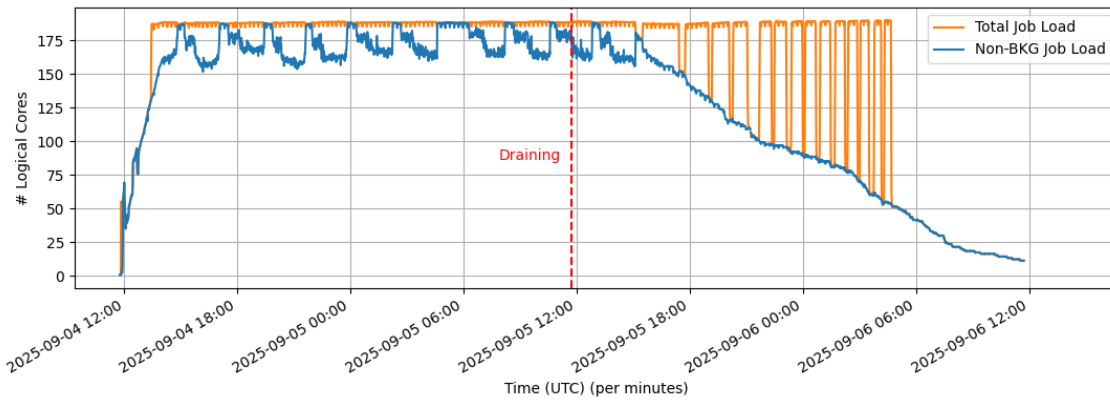


Figure 5.6: Full CPUs utilised on drones with background jobs. Before draining all 192 CPUs are utilised. When draining starts, foreground jobs gradually vacate the drone, as they finish, however the CPUs used still remains almost full thanks to the background jobs. Before draining it can be seen that foreground jobs have periodic declines, which is background job interference into foreground jobs. The interference range is ~ 0 -20 CPUs (~ 0 -10%), which is 5% on average observed CPU efficiency decline. Here, each "pillar" and each dip period in CPU usage before draining is a single background job. As there is more and more free CPUs due to foreground jobs finishing after draining, the pillars get narrower, the background jobs can use more CPUs and get faster.

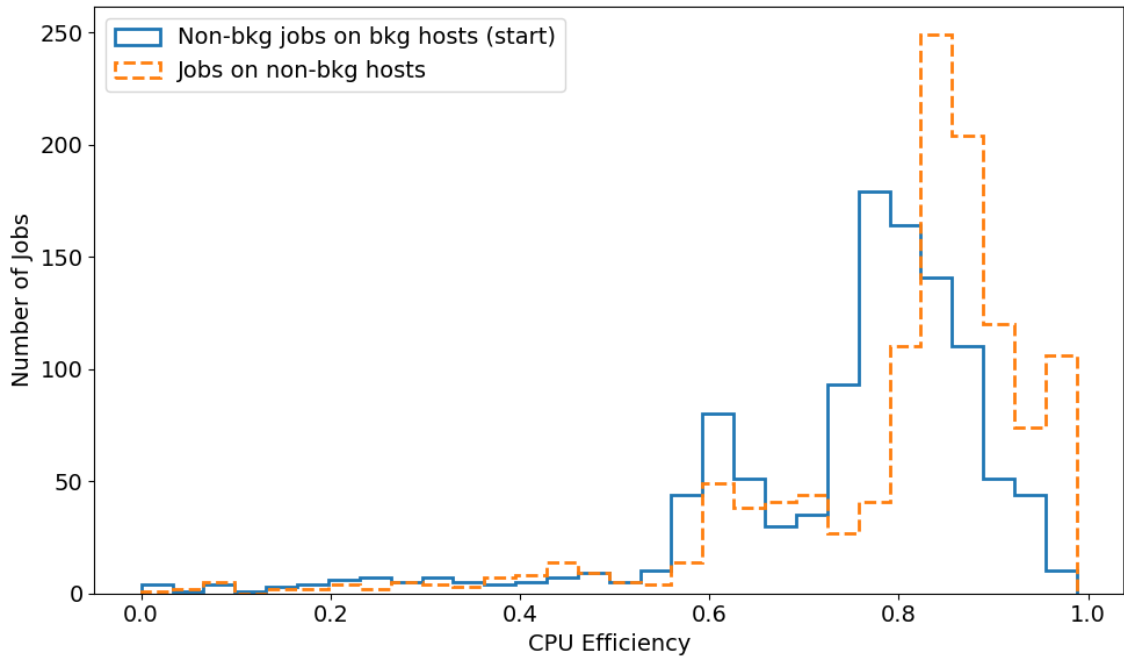


Figure 5.7: CPU efficiency distribution of foreground jobs. Foreground jobs on nodes without background jobs (orange, dashed) and with background jobs (blue, solid) CPU efficiency is calculated as relation of CPU time to wall time multiplied by number of CPUs registered in PanDA.

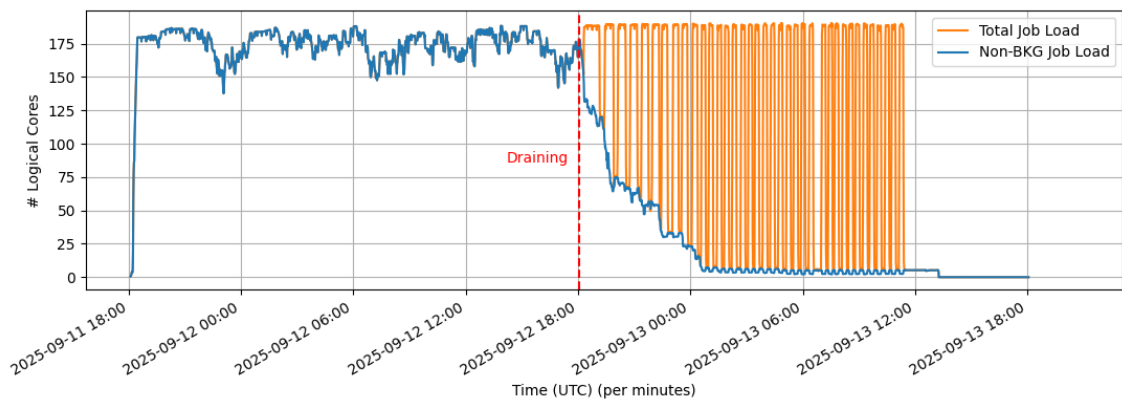


Figure 5.8: Third experiment. Background jobs were activated only after Draining phase started. Although this eliminated background job interference into foreground jobs' CPU usage, the experiment revealed that foreground jobs do not occupy all CPUs before draining.

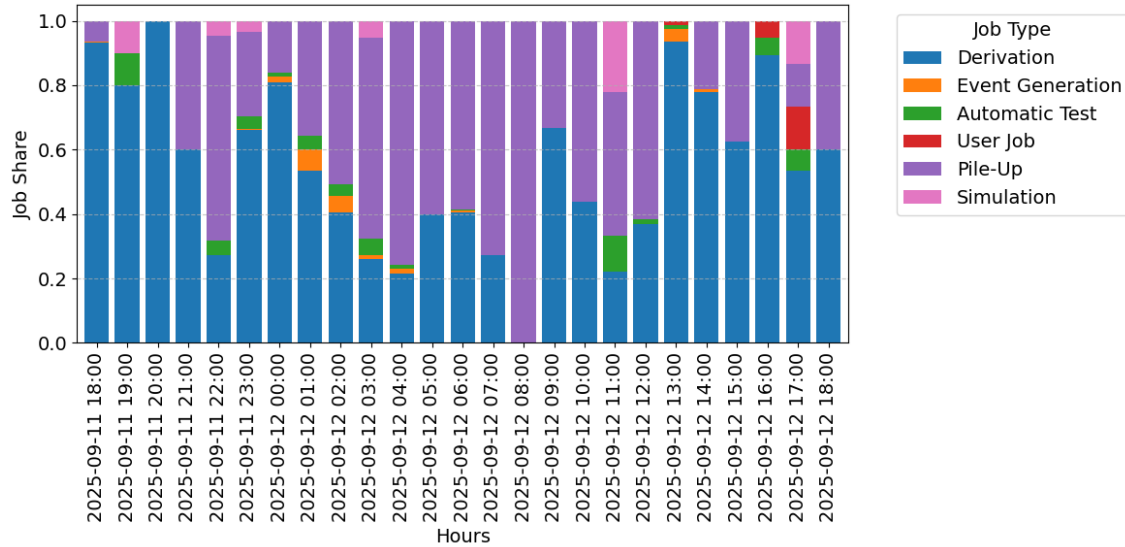


Figure 5.9: Job mix of foreground jobs with background jobs on the node for the third experiment. Job share significantly differs from the first 2 experiments (Fig. 5.4). This experiment has an overwhelming presence of predictable production jobs and minimal amount of user jobs.

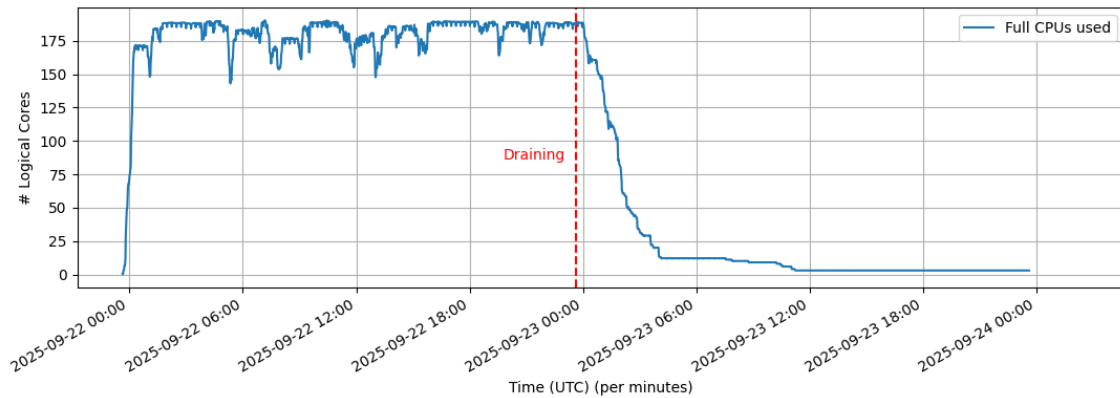


Figure 5.10: Experiment 4. Nodes without background jobs. Foreground jobs, as in the third experiment do not consume all available CPUs before draining. The gaps are present, however they are not as significant.

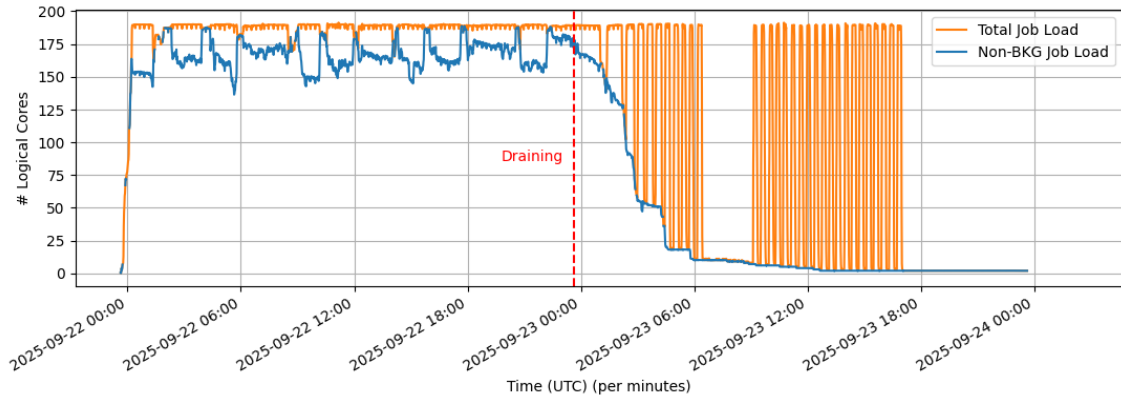


Figure 5.11: Experiment 4. Nodes with background jobs launched from the start. The foreground jobs do not occupy all available CPUs before draining. Background jobs fill the gaps. However, background jobs still interfere with the foreground jobs and the downs in CPU usage of foreground jobs are more significant due to this factor.

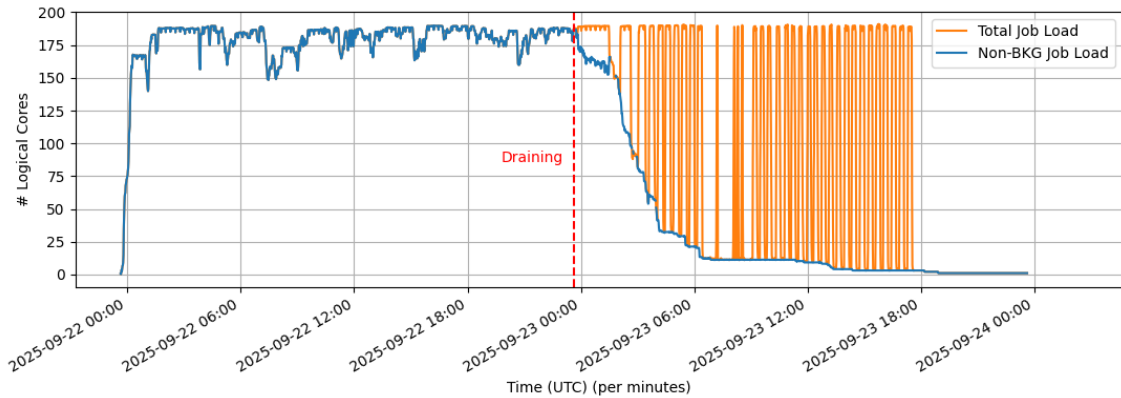


Figure 5.12: Experiment 4. Nodes with background jobs launched starting from draining. Before draining foreground jobs do not occupy all available CPUs, even though the usage is close to maximum (192 CPUs). After draining background jobs use CPUs to maximum.

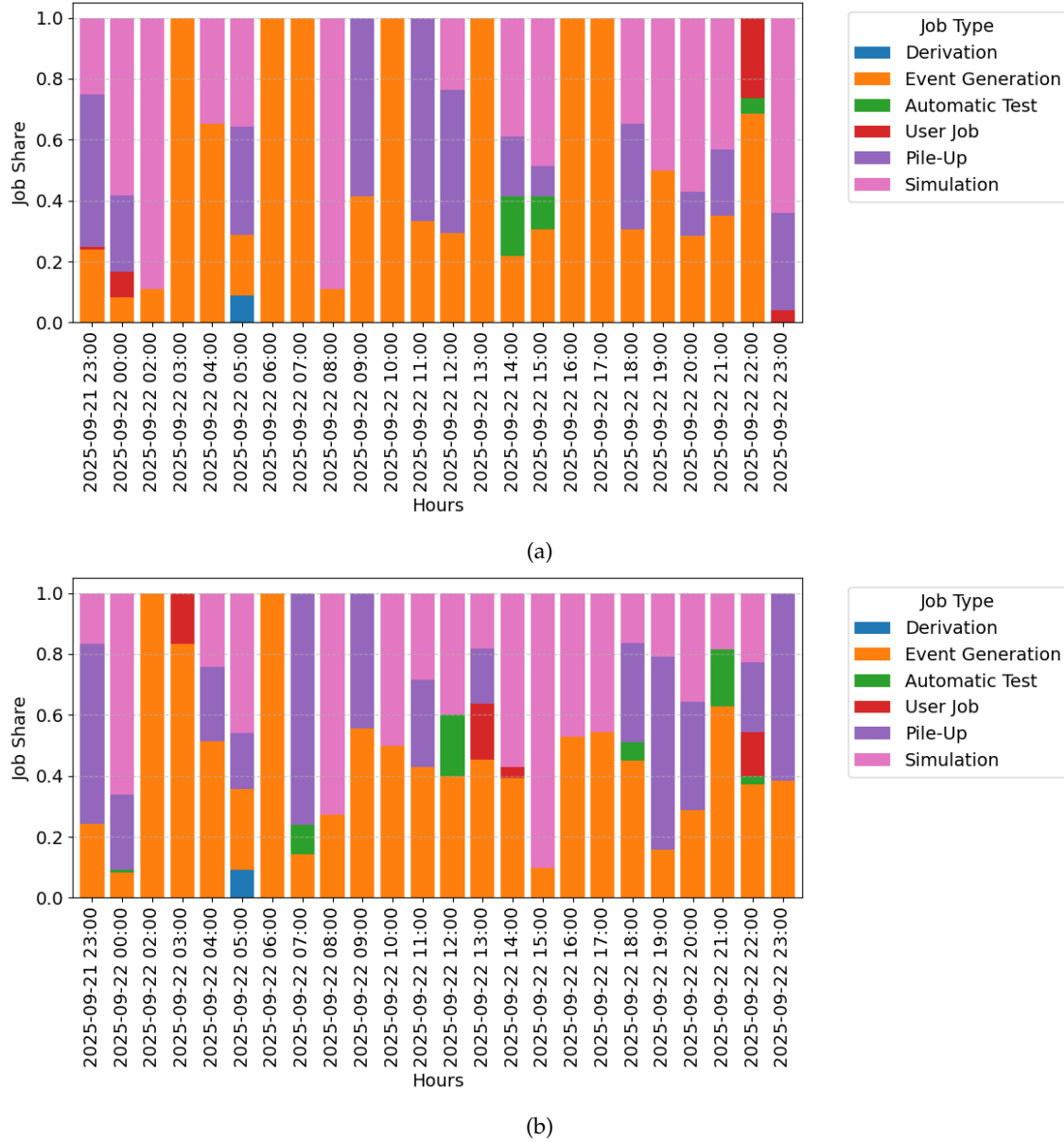


Figure 5.13: Experiment 4. Job mix of foreground jobs on nodes with background jobs launched from start (a) and with background jobs launched only during draining (b). Job share significantly differs from the first 2 experiments (Fig. 5.4). This experiment has an overwhelming presence of predictable production jobs and minimal amount of user jobs.

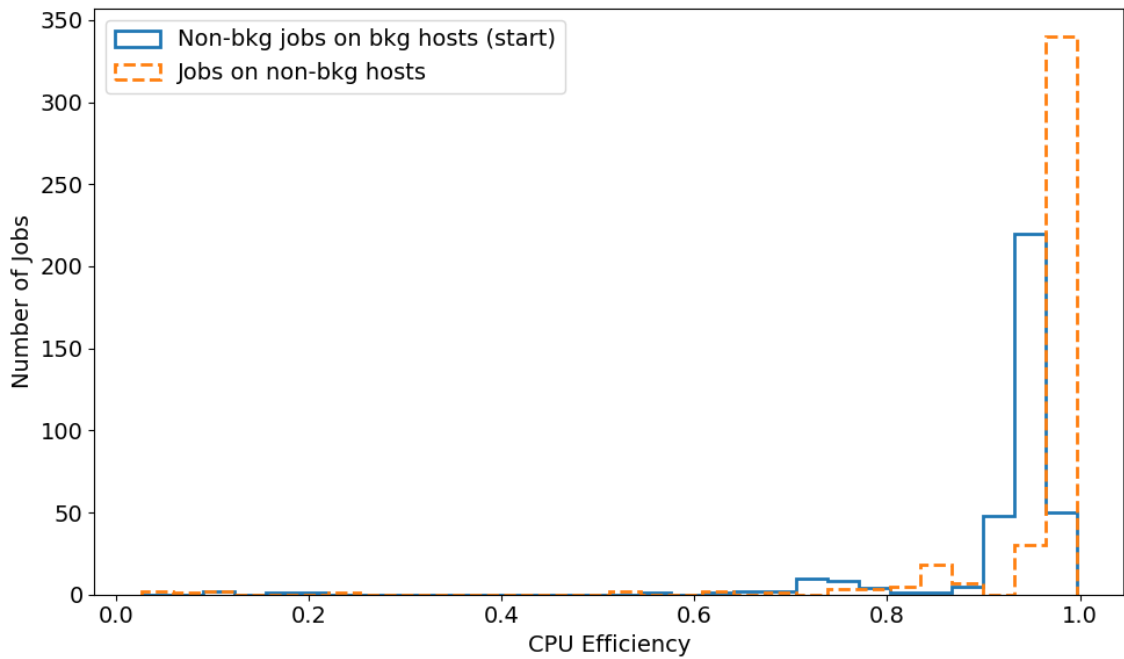


Figure 5.14: CPU efficiency distribution of foreground jobs on nodes with (blue, solid) and without (orange, dashed) background jobs. Background jobs launched from the start.

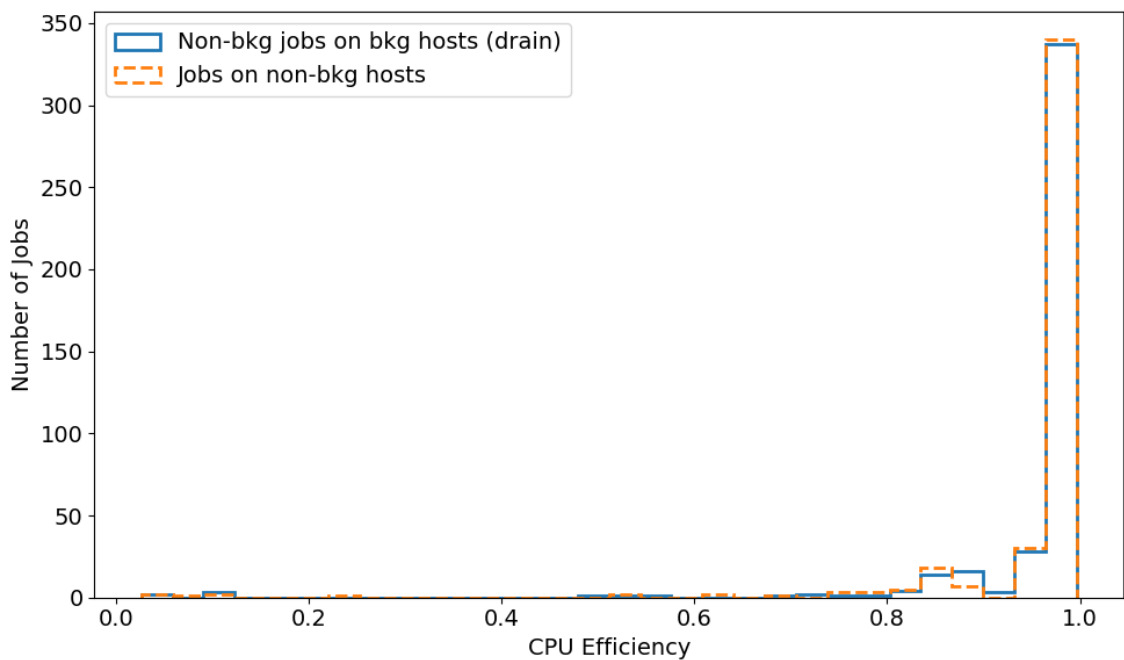


Figure 5.15: CPU efficiency distribution of foreground jobs on nodes with (blue, solid) and without (orange, dashed) background jobs. Background jobs launched only during draining.

5.3 Codebase Improvements

The refactoring of the drone management framework brought a number of practical benefits in terms of maintainability, configurability and deployment reliability. While the core functionality of launching drones as virtual worker nodes on the NHR cluster remained unchanged, the restructured implementation adopted a modular and configuration-driven design that improved day-to-day usability and long-term extensibility.

From a maintainability perspective, the monolithic scripts used in the initial implementation were decomposed into a collection of helpers organised under the `common/helpers` namespace. Subsystems such as configuration parsing, container orchestration, cgroup handling and working directory setup were isolated into dedicated modules. This eliminated duplicated logic across scripts and reduced the cognitive overhead for new contributors. Each component now serves a clearly defined purpose, allowing changes to be made locally without unintended side effects.

Configurability was enhanced by introducing a YAML-based configuration layer. Parameters such as container definition files, binding rules, subdirectory layout and cgroup parent paths are expressed declaratively in `default.yml`. Runtime placeholders (`<paths.>` and `<subdirectories.>`) allow configuration files to remain portable across different deployment environments. This design makes it possible to adjust drone behaviour by editing a single configuration file, avoiding the need for code modifications.

Deployment usability also improved with the introduction of a single entry point, `drone.py`, which provides subcommands for installation and submission. Instead of relying on multiple shell scripts with implicit dependencies, drones are now deployed using explicit commands:

```
./drone.py install --config drone_configs/default.yml
./drone.py submit --config drone_configs/default.yml --lifetime 604800
```

The `install` stage prepares the working directory and builds the container, while `submit` handles deployment, CVMFS setup, cgroup configuration and container startup. The separation of these stages mirrors the lifecycle of drone management and simplifies troubleshooting.

The design of `container.py` allows container flags and binding options to be added or removed through configuration without editing Python code. Similarly, `cgroups.py` resolves parent cgroups dynamically from environment variables such as `SLURM_JOB_ID`, enabling adaptation to different batch systems with minimal changes. These design choices future-proof the codebase, ensuring it can accommodate changing site requirements.

Reliability and debuggability improved due to stricter path handling and environment setup. Functions in `workdir.py` enforce that all operations remain confined to the intended output directory, reducing the risk of accidental overwrites. The initialisation of the Apptainer environment is now explicit and consistent across deployments and error messages during container startup

provide clear diagnostics when misconfigurations occur.

The refactoring did not alter the high-level logic of drone lifecycle management but transformed the codebase into a maintainable, modular and easy to read and document tool. This aligns with the methodological goals described in Chapter 4 and eases the work for integration of Tier-2 drones in heterogeneous HPC environments.

In addition to qualitative improvements, the refactoring introduced measurable gains in code maintainability and usability. The original framework consisted of over 1100 lines of mixed-purpose scripts with duplicated logic, whereas the refactored version reduced duplication by consolidating helpers into fewer than 600 lines spread across dedicated modules. Operational complexity decreased from four separate scripts to a single unified entry point, reducing deployment to two explicit commands. Over 30 configuration parameters that were previously hardcoded are now expressed declaratively in YAML, improving portability across environments. Furthermore, input validation and path confinement added in the refactoring increased the number of explicit error checks from 2 to 11, strengthening reliability. Finally, the introduction of comprehensive documentation and modular functions raises the fraction of documented components from under 10% to nearly 100%, improving accessibility for future developers.

Chapter 6

Discussion and Conclusion

6.1 Discussion

This section discusses the results presented in Chapter 5 in the context of the design and methodology developed in Chapter 4. The focus is on how the architectural choices influenced the outcomes, what the monitoring and experiments revealed and what lessons can be drawn for production operation.

The project is intended to integrate opportunistic HPC resources into the WLCG computing model via HTCondor drones, while ensuring strict separation of foreground and background workloads. A central requirement was to maximise utilisation of the booked CPUs and to provide end-to-end observability through an ELK-based monitoring pipeline.

The choice to place Condor readers centrally on the monitoring node rather than deploying per-node agents proved effective. Scheduler-level queries already aggregate information on slot states and job lifecycles, eliminating redundancy and reducing deployment overhead. The trade-off is a potential dependency on scheduler responsiveness and parsing stability, since the reader relies on textual output of `condor_status`.

Similarly, the decision to transport data via file-based JSON logs shipped by Filebeat provided resilience and debuggability. Agents remain functional during temporary outages of Elasticsearch or Logstash and raw logs are preserved for inspection. The drawback is added disk I/O and a small delay before data reaches dashboards, but this was acceptable for the intended monitoring granularity.

The use of cgroup-based isolation was essential for interference limitation and for evaluating interference between foreground and background jobs. While cgroups allowed accurate tracking of slot usage, they also revealed a limitation: single-core background jobs, scheduled with low weights, still competed with single-core foreground jobs. This behaviour explains the observed 5%

efficiency loss in foreground workloads when background jobs were enabled from launch.

The dashboards confirm that the monitoring design achieved comprehensive observability. Network counters at the worker-node level allowed inspection of both aggregate and per-interface traffic, useful for diagnosing bottlenecks in data-intensive HEP workflows. Power metrics, collected centrally via IPMI, provided insight into the energy footprint of workloads and enabled correlation between utilisation and consumption. Lifecycle views, distinguishing alive and draining nodes, validated that the drones retire jobs as expected under walltime constraints. Finally, the display of PanDA queue flags confirmed that jobs were consistently matched to their designated queues, ensuring policy compliance.

The experiments demonstrated both the promise and the risks of background job execution. Background tasks successfully filled idle resources during draining, recovering otherwise wasted capacity. However, when launched from the start, they interfered with foreground single-core jobs due to Condor's slot scheduling and cgroup weight settings. The average 5% efficiency loss observed across experiments shows the importance of restricting background work to periods where it cannot impact production jobs.

A secondary finding was that foreground jobs do not always saturate all available CPUs before draining. In some experiments, job mixes left gaps even without background jobs, highlighting that utilisation patterns depend strongly on workload composition. This suggests that policies for activating background slots may need to adapt dynamically to job mix conditions.

The refactored codebase significantly improved maintainability and reproducibility of the deployment. Modular helpers for configuration, containers, cgroups and workdir management reduced duplication and made behaviour explicit. The introduction of a YAML-based configuration file provided a single source of truth for parameters such as paths, cgroup templates, PanDA flags and container bindings. A unified CLI interface (`drone.py`) simplified operations by exposing only two commands (`install` and `submit`), each with well-defined arguments. These changes ensured that experiments could be repeated consistently and reduced the likelihood of operator error in complex environments.

6.2 Limitations

While the project successfully demonstrated the feasibility of integrating opportunistic HPC resources through drones and provided comprehensive monitoring, several limitations were observed that constrain applicability in production-scale deployments.

The behaviour of background jobs was shown to depend strongly on the mix of foreground jobs. In some experiments, foreground jobs fully saturated resources before draining, leaving no safe capacity for background tasks. In others, gaps were visible even without background jobs. This variability means that fixed policies for background job activation are suboptimal. A production-

ready system would require adaptive strategies, potentially informed by real-time monitoring of slot utilisation and job efficiency.

Although cgroups ensured correct accounting and isolation, interference between single-core background and foreground jobs was unavoidable due to how Condor assigns weights within multicore slots. Even with low weights, background jobs occasionally delayed foreground tasks, resulting in the measured 5% efficiency drop. Mitigating this effect would require either fine-grained scheduler tuning or container-level resource throttling, neither of which was implemented during this project.

The file-based logging approach provided resilience but introduced latency, with metrics only becoming visible after Filebeat shipped them to Logstash and Elasticsearch. For interactive debugging or short-lived jobs, this delay reduces usefulness. Furthermore, although agent overhead was minimal, log I/O could grow significantly under very large-scale deployments, requiring careful tuning of buffer sizes and retention policies.

The refactored codebase emphasised clarity and reproducibility but did not yet optimise for large-scale automation or fault tolerance. For example, the lifetime of drones was specified statically through configuration and failed drones required manual resubmission. Similarly, while configuration was centralised in YAML, validation of inputs and error handling remained limited. These factors restrict scalability without further development. However, the automation of this process is in the plan and will be realised via the Opportunistic Balancing Daemon (COBalD)/Transparent Adaptive Resource Dynamic Integration System (TARDIS) software [53].

6.3 Future Work

Several directions for future work have been identified to advance the system beyond the current state.

One key improvement is the development of adaptive scheduling policies for background jobs. Instead of relying on fixed activation rules, the system could leverage real-time monitoring of slot utilisation, job efficiency and draining behaviour to dynamically enable or disable background slots. Such feedback-driven control would mitigate the risk of interfering with foreground jobs while maximising the utilisation of idle resources.

The observed interference between single-core background and foreground jobs highlights the need for finer-grained resource allocation. Future work may investigate enhanced cgroup isolation, such as `cpu.shares` or `cpu.max`, to throttle background job execution more aggressively when foreground demand is high. Alternatively, scheduler-level policies in HTCondor could be tuned to reduce contention by differentiating background tasks with explicit priorities or slot types.

Currently, drones are launched manually with static configuration parameters. A logical next step is full automation of drone lifecycle management, including automatic instantiation, monitoring

and retirement of drones based on demand. The COBalD/TARDIS framework provides an existing foundation for this functionality and integration with the refactored codebase is already foreseen. This would enable a production system where drones elastically scale to match the workload while maintaining strict queue separation.

While the ELK stack proved effective for resilient metric storage and visualisation, its batch-oriented nature limits responsiveness. Future enhancements could include near-real-time streaming pipelines for metrics requiring interactive feedback. Such an addition would complement the existing architecture by providing low-latency views for debugging, while Elasticsearch remains the authoritative backend for long-term analysis.

These directions point towards transforming the current work into a more advanced, scalable and autonomous system for opportunistic use of HPC resources. By combining adaptive background job policies, stronger isolation mechanisms, automation through COBalD/TARDIS and enhanced monitoring, the system could significantly increase the efficiency and flexibility of resource usage in hybrid grid-HPC environments.

6.4 Conclusion

This work demonstrated the feasibility of integrating opportunistic background jobs to maximise the use of HPC resources in the WLCG ecosystem using containerised drones, supported by a comprehensive monitoring stack and improved codebase. The monitoring pipelines based on lightweight agents and the ELK stack provided detailed visibility into network traffic, power consumption, node lifecycle states and queue assignments, enabling both operational oversight and experimental validation.

Through controlled experiments, the role of background jobs was systematically evaluated. Results showed that background tasks can increase overall CPU utilisation, particularly during draining phases, but that interference with single-core foreground jobs occurs if they are enabled prematurely. These findings underline the importance of adaptive policies that align background execution with workload characteristics.

The refactoring of the drone management codebase introduced a structured, modular design with centralised configuration, clearer abstractions and reproducible deployment procedures. While not yet optimised for automation or large-scale fault tolerance, this foundation enables future integration with orchestration frameworks such as COBalD/TARDIS.

Taken together, the project provides both a methodological framework and a practical prototype for leveraging HPC resources in hybrid computing environments. By combining lightweight monitoring, controlled opportunistic scheduling and modular code, it establishes a path toward production-ready systems that enhance the efficiency and flexibility of large-scale scientific workflows.

Bibliography

- [1] Lyndon Evans, Lyn Evans, *The Large Hadron Collider*, Fundamental sciences. Physics, EPFL Press, Lausanne (2009)
- [2] The ATLAS Collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST **3(08)**, S08003–S08003 (2008)
- [3] I. Bird, *Computing for the Large Hadron Collider*, Ann. Rev. Nucl. Part. Sci. **61(1)**, 99–118 (2011)
- [4] The ATLAS Collaboration, *Software and computing for Run 3 of the ATLAS experiment at the LHC*, Eur. Phys. J. C **85**, 234 (2025)
- [5] T. Berners-Lee, *Information Management: A Proposal*, Technical Report CERN-DD-89-001, CERN (1989)
- [6] T. Berners-Lee, et al., *World-Wide Web: The Information Universe*, Electronic Networking: Research, Applications and Policy **1(2)**, 52 (1992)
- [7] CERN Yellow Reports: Monographs, *CERN Yellow Reports: Monographs, Vol. 10 (2020): High-Luminosity Large Hadron Collider (HL-LHC): Technical design report* (2020)
- [8] The ATLAS Collaboration, *Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC*, Phys. Lett. B **716(1)**, 1 (2012)
- [9] U. S. Polisetty, et al. (ATLAS), *Integration of the Goettingen HPC cluster Emmy to the WLCG Tier-2 centre GoeGrid and performance tests*, Technical report, CERN, Geneva (2025)
- [10] The ATLAS Collaboration (ATLAS), *ATLAS Distributed Computing: Its Central Services core*, EPJ Web Conf. **214**, 03061 (2019)
- [11] The ATLAS Collaboration, *ATLAS Distributed Computing Operation Shift Teams experience during the discovery year and beginning of the Long Shutdown 1*, J. Phys. Conf. Ser. **513(3)**, 032085 (2014)
- [12] M. Turilli, M. Santcroos, S. Jha, *A Comprehensive Perspective on Pilot-Job Systems*, ACM Comput. Surv. **51(2)** (2018)

- [13] F. Stagni, A. Valassi, V. Romanovski, *Integrating LHCb workflows on HPC resources: status and strategies*, EPJ Web Conf. **245**, 09002 (2020)
- [14] M. Belkin, et al., *Container solutions for HPC Systems: A Case Study of Using Shifter on Blue Waters*, in *Proceedings of the Practice and Experience on Advanced Research Computing: Seamless Creativity*, PEARC '18, Association for Computing Machinery, New York, NY, USA (2018)
- [15] The ATLAS Collaboration, *Harvester: an edge service harvesting heterogeneous resources for ATLAS*, EPJ Web Conf. **214**, 03030 (2019)
- [16] P. Buncic, et al., *CernVM: Minimal maintenance approach to virtualization*, J. Phys. Conf. Ser. **331(5)**, 052004 (2011)
- [17] J. Blomer, et al., *Distributing LHC application software and conditions databases using the CernVM file system*, J. Phys. Conf. Ser. **331(4)**, 042003 (2011)
- [18] J. Blomer, et al., *Micro-CernVM: slashing the cost of building and deploying virtual machines*, J. Phys. Conf. Ser. **513(3)**, 032009 (2014)
- [19] J. Blomer, et al., *New directions in the CernVM file system*, J. Phys. Conf. Ser. **898(6)**, 062031 (2017)
- [20] R. Popescu, J. Blomer, G. Ganis, *Towards a responsive CernVM-FS architecture*, EPJ Web Conf. **214**, 03036 (2019)
- [21] L. Promberger, J. Blomer, V. Völkl, M. Harvey, *CernVM-FS at Extreme Scales*, EPJ Web of Conf. **295**, 04012 (2024)
- [22] G. M. Kurtzer, V. Sochat, M. W. Bauer, *Singularity: Scientific containers for mobility of compute*, PLoS ONE **12(5)**, 1 (2017)
- [23] G. Roy, et al., *Evaluation of containers as a virtualisation alternative for HEP workloads*, J. Phys. Conf. Ser. **664(2)**, 022034 (2015)
- [24] M. Storetvedt, et al., *Grid services in a box: container management in ALICE*, EPJ Web Conf. **214**, 07018 (2019)
- [25] C. Acosta-Silva, et al., *Exploitation of network-segregated CPU resources in CMS*, EPJ Web Conf. **251**, 02020 (2021)
- [26] B. Tovar, et al., *Harnessing HPC resources for CMS jobs using a Virtual Private Network*, EPJ Web Conf. **251**, 02032 (2021)
- [27] V. Amoiridis, et al., *The CMS Orbit Builder for the HL-LHC at CERN*, EPJ Web of Conf. **295**, 02011 (2024)
- [28] A. Aimar, et al., *Unified Monitoring Architecture for IT and Grid Services*, J. Phys. Conf. Ser. **898(9)**, 092033 (2017)

- [29] A. Aimar, et al., *MONIT: Monitoring the CERN Data Centres and the WLCG Infrastructure*, EPJ Web Conf. **214**, 08031 (2019)
- [30] The ATLAS Collaboration, *ATLAS BigPanDA monitoring*, J. Phys. Conf. Ser. **1085(3)**, 032043 (2018)
- [31] The ATLAS Collaboration (ATLAS), *The BigPanDA self-monitoring alarm system for ATLAS*, Technical report, CERN, Geneva (2018)
- [32] The ATLAS Collaboration, *Enhancements in Functionality of the Interactive Visual Explorer for ATLAS Computing Metadata*, EPJ Web Conf. **245**, 05032 (2020)
- [33] The CMS Collaboration, *The evolution of the CMS monitoring infrastructure*, EPJ Web Conf. **251**, 02004 (2021)
- [34] C. Ariza-Porras, V. Kuznetsov, F. Legger, *The CMS monitoring infrastructure and applications*, Comput. Softw. Big Sci. **5(1)** (2021)
- [35] J. A. Lopez-Perez, et al., *The web based monitoring project at the CMS experiment*, J. Phys. Conf. Ser. **898(9)**, 092040 (2017)
- [36] P. Andrade, et al., *WLCG Dashboards with Unified Monitoring*, EPJ Web Conf. **245**, 07049 (2020)
- [37] A. Mu'alem, D. Feitelson, *Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling*, IEEE Trans. Parallel Distrib. Syst. **12(6)**, 529 (2001)
- [38] A. B. Yoo, M. A. Jette, M. Grondona, *SLURM: Simple Linux Utility for Resource Management*, in *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
- [39] M. Fischer, et al., *Lightweight dynamic integration of opportunistic resources*, EPJ Web Conf. **245**, 07040 (2020)
- [40] O. Freyermuth, et al., *Operating an HPC/HTC Cluster with Fully Containerized Jobs Using HTCondor, Singularity, CephFS and CVMFS*, Comput. Softw. Big Sci. **5(1)** (2021)
- [41] W. Wu, D. Cameron, D. Qing, *Using ATLAS@Home to Exploit Extra CPU from Busy Grid Sites*, Comput. Softw. Big Sci. **3(1)** (2019)
- [42] A. F. Boyer, et al., *DIRAC Site Director: Improving Pilot-Job provisioning on grid resources*, Future Gener. Comput. Syst. **133**, 23 (2022)
- [43] W. S. Cleveland, *Robust Locally Weighted Regression and Smoothing Scatterplots*, J. Am. Stat. Assoc. **74(368)**, 829 (1979)
- [44] B. J. Tang, et al., *The MIT Supercloud Workload Classification Challenge*, in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 708–714 (2022)

- [45] Google SRE, *Monitoring Distributed Systems* (2016), in *Site Reliability Engineering: How Google Runs Production Systems*, Google SRE Book
- [46] S. Schlagkamp, et al., *Understanding User Behavior: From HPC to HTC*, *Procedia Comput. Sci.* **80**, 107 (2016)
- [47] V. Reis, D. Glesser, D. Trystram, *Improving Backfilling by Using Machine Learning to Predict Running Times*, in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 275–278 (2015)
- [48] Amazon Web Services, *Amazon Simple Storage Service (S3)* (2024), AWS Documentation
- [49] CS3Org, *Reva: Interoperable Cloud Storage Framework* (2024), software project maintained by CS3Org
- [50] CS3MESH4EOSC Consortium, *ScienceMesh: Interoperable Platform for Collaborative Research and Education* (2024)
- [51] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Security in Distributed Systems* (2020), lecture notes from *Operating Systems: Three Easy Pieces*, University of Wisconsin–Madison.
- [52] D. Thain, T. Tannenbaum, M. Livny, *Distributed computing in practice: the Condor experience*, *Concurr. Comp.-Pract. E.* **17(2-4)**, 323 (2005)
- [53] M. Böhler, et al., *Transparent Integration of Opportunistic Resources into the WLCG Compute Infrastructure*, *EPJ Web Conf.* **251**, 02039 (2021)

Appendix

Description of ATLAS Job Types

The ATLAS experiment utilizes a variety of job types to simulate, reconstruct and analyse data collected from the LHC. Each job type corresponds to a specific stage in the data processing pipeline and understanding their individual resource characteristics is essential for performance modeling and infrastructure planning. The descriptions below are based on the 2024 ATLAS HL-LHC Computing Model [4] and production conventions observed in this study.

evgen – Event Generation `evgen` jobs generate simulated particle collisions using Monte Carlo techniques. These represent possible physics events as modeled by theoretical predictions. `evgen` jobs are CPU-bound with minimal I/O and form the first stage of Monte Carlo production. They produce HepMC-format records for subsequent simulation.

simul – Simulation `simul` jobs process generated events to simulate their interaction with the ATLAS detector, using Geant4. These jobs consume large CPU resources and generate raw hit-level data that mimics the response of the actual detector systems. Simulation is one of the most resource-intensive stages in the workflow.

pile – Pile-up Simulation `pile` jobs simulate multiple proton-proton interactions occurring in the same bunch crossing (pile-up). They overlay minimum-bias events onto signal events to reflect realistic HL-LHC conditions. These jobs require high memory and perform significant file merging operations.

recon – Reconstruction `recon` jobs convert raw detector (or simulated) data into high-level physics objects such as tracks and jets. They are both CPU and I/O intensive and depend on detector calibrations and alignment. Reconstruction is performed on both real and simulated data.

reprocessing `reprocessing` jobs re-run reconstruction using improved software, updated detector conditions or revised calibrations. They are essential for producing consistent datasets across time and have similar resource requirements to standard `recon` jobs.

deriv – Derivation `deriv` jobs apply selections, slimming and skimming to reduce full reconstructed datasets into smaller, analysis-friendly formats. This stage is crucial for managing HL-LHC-scale data volumes and ensuring efficiency for physics groups. These jobs are moderately I/O intensive.

pmerge – Pile-up Merge `pmerge` jobs merge pile-up events with hard-scatter simulation outputs, forming combined datasets before reconstruction. These jobs require high I/O throughput and careful synchronization of event timing metadata.

eventIndex `eventIndex` jobs extract metadata from processed events and build searchable catalogs. These indexes allow efficient event lookup across distributed datasets. While rare, these jobs often involve large input reads and are I/O bound.

gangarobot `gangarobot` jobs are part of the automated validation infrastructure. They regularly test site availability, environment configuration and software compatibility. These are short jobs with light resource footprints but are critical for operational monitoring.

panda-client `panda-client` jobs represent direct user submissions via the PanDA client. They include a wide range of workflows, including custom simulations, private analyses or software validation. Resource demands vary greatly and depend on the specific user workload.

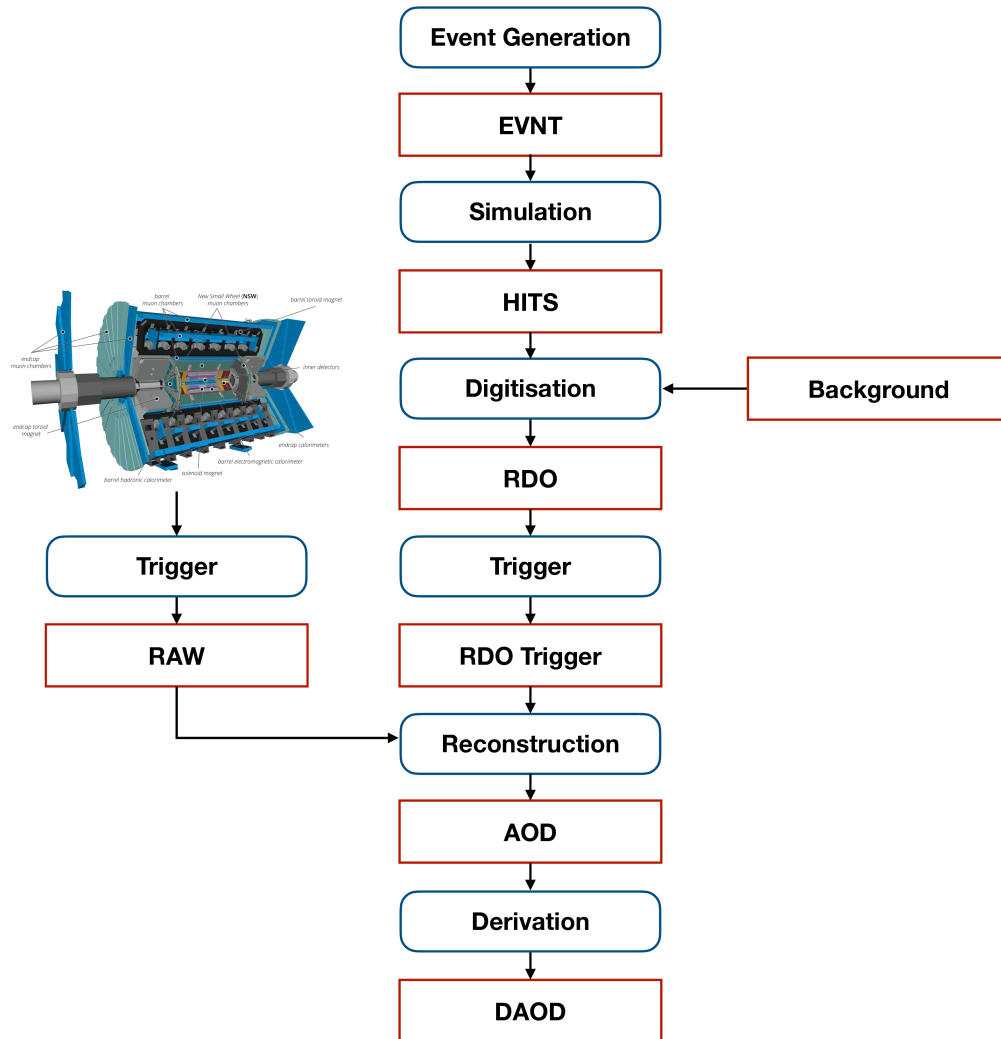


Figure 1: The standard software workflow of the ATLAS experiment. Processing steps are represented by blue ovals, with output formats represented as red boxes. The various steps and data formats are described in the text. The background entering digitisation may be additional simulated HITS files, pre-digitised RDO files or specially processed RAW detector data. [4]

