Enhancing Grid Middleware Performance

Priority Management and Locking Time optimization in JAliEn

Jørn-Are Flaten

Master's thesis in Software Engineering at

Department of Computer science, Electrical engineering and Mathematical sciences,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

November 2024

Western Norway
University of
Applied Sciences

Abstract

This thesis presents work that aimed at enhancing the fairness of the user scheduling algorithm and improve system performance by optimizing complex database queries in the Java ALICE Environment (JAliEn), a Grid middleware at CERN with access to the Worldwide LHC Computing Grid (WLCG). To address these challenges, the necessary parameters for a novel user scheduling algorithm were identified, and the algorithm was implemented to ensure fairness for both high- and low-priority users while minimizing system impact. The complex database queries caused prolonged locking times, blocking other processes from accessing the database. These queries were optimized by breaking down the complex queries to simpler ones and updating individual rows while avoiding read locks. This change significantly reduced the duration of the longest lock and allowed other processes to access the database between queries. The total locking duration has been decreased from over 13 seconds to less than 900 milliseconds, while allowing processes to access the database between each write lock. The resulting code has been deployed in production and is used by researchers worldwide to analyze data from the ALICE experiment at CERN. The system is now more efficient and fairer than before and the resulting code plays a crucial role in the job management system within the grid middleware JAliEn, which processes over a million batch jobs daily.

Acknowledgements

First and foremost, I would like to thank my supervisors Bjarte Kileng and Håvard Helstrup for their guidance, feedback and support throughout the project. Additionally, I would like to thank the experts in the JAliEn at CERN for their valuable input and feedback. Specifically, I would like to thank Costin Grigoras for sharing his expertise, feedback and for being an invaluable resource and guide in understanding the JAliEn system and the requirements of this project. I'm thrilled and very grateful to have been able to contribute to such an interesting and important project at CERN.

Contents

Listings

List of Figures

List of Tables

Acronyms

ACID Atomicity, Consistency, Isolation, and Durability.

ALICE A Large Ion Collider Experiment.

AliEn ALICE Environment - A legacy grid middleware previously used by the ALICE experiment at CERN.

ATLAS A Toroidal LHC ApparatuS.

CERN European Organization for Nuclear Research (Conseil Européen pour la Recherche Nucléaire).

CPU Central Processing Unit.

GPU Graphics Processing Unit.

HTCondor A batch workload management system.

I/O Input / Output.

JAliEn Java ALICE Environment - the grid middleware used by the ALICE experiment at CERN.

Java NIO Java library which defines buffers, which are containers for data, and provides an overview of the other NIO packages.

JDL Job Description Language.

JVM Java Virtual Machine.

LHC Large Hadron Collider.

optimizer A Java class running on central machines, asynchronously, that attempts to consolidate, synchronize and update information.

TCP Transmission Control Protocol.

TCP/IP Transmission Control Protocol / Internet Protocol.

WLCG Worldwide LHC Computing Grid.

Chapter 1

Introduction

I have no special talent. I am only passionately curious.

Albert Einstein

1.1 Background

The computing power of modern computers has increased significantly over the last few decades, as has the amount of data generated. The processing power of the small computer in your pocket is thousands of times more powerful than the computers that sent the first astronauts to the moon. Today's computers can solve multiple complex problems simultaneously and can be combined to form clusters to tackle even larger challenges. One particular branch of computing is distributed computing, where multiple computers collaborate to solve a single task. These distributed environments often consist of heterogeneous systems that utilize geographically dispersed computing resources to process large amounts of data over extended periods. A large-scale variant of this is Grid computing, which is particularly suited for scientific and research purposes, such as CERN's Large Hadron Collider computing Grid. Such environments are often referred to as high-throughput computing, as they focus on maximizing the total amount of computational work completed over time, rather than the speed of individual tasks [1].

The A Large Ion Collider Experiment (ALICE) [2] at CERN is, together

with ATLAS, CMS, LHCb, one of the four large particle experiments at the Large Hadron Collider (LHC). The detector is a heavy-ion detector using proton-proton and lead-lead collisions to study the quark-gluon plasma[2], a state of matter that existed shortly after the Big Bang.

The case for this project is the Java ALICE Environment (JAliEn), which is a Grid middleware that provides an interface to access a subset of the LHC worldwide computing Grid (WLCG) [3]. This system is used to process huge amounts of data which is generated by the ALICE experiment. Data produced by the experiment is initially stored locally in a 130 PB buffer, and filled at a rate of 120 GB/s after compression at lead-lead collisions [4]. ALICE's computing requirements are satisfied by the institutes participating in the collaboration. The 60 computing centers contribute heterogeneous resources, assembling more than 200,000 CPU cores and 400 PB of disk and tape storage [4]. JAliEn aims to provide a unified view for users to access the Grid resources, regardless of the environment it runs on. Further on this in Section 2.2.4 (p. 35).

1.2 Problem statement

As a result of the hardware upgrade to the ALICE experiment for LHC Run 3 [3], it became necessary to upgrade the software to manage job scheduling and handling. The massively increased data generation from the experiment requires the Grid middleware to handle significantly more CPU and resources for storage [3]. To offer horizontal scaling [3] and support multi-core workloads core processing capabilities [5] a new Grid middleware was designed, the JAliEn became the successor of the ALICE Environment (AliEn) Grid middleware. Most of the functionality had been ported from AliEn to JAliEn by the year 2023, however the old bookkeeping service still remained. The bookkeeping service is responsible for keeping track of the job status, resource usage, and user priorities. A job goes through multiple states when it is submitted until it reaches a final state which might be successful or a type of erroneous state. The priority of a user is used to determine who should be given the chance to execute their job next when resources are available. Resources used such as CPU time and the number of allocated CPU cores are tracked by the bookkeeping service though a suite of optimizers. An optimizer is a Java class that runs on the central machines, asynchronously,

to consolidate, synchronize and update information.

The bookkeeping service is a central service in the JAliEn Grid middleware, and interacts with multiple tables in the central database for read and write operations. Partially, these operations include updating resource usage information and current user priorities, summarizing resource usage per site and for every type of job state per user. The main responsibility for the service is to ensure that information is synchronized between sites and the central services through a common database. The main problem with the legacy implementation of this service is that it involves complex queries that locks the central database for long durations of time when updating values, due to the large number of join and update operations it performs. The combination of these queries with other complex queries that interact with the some of the same tables can cause periods of long read and write locks on the database, resulting in a situation where other processes are unable to access the database. These situations are difficult to recover from and can cause the database to be unresponsive for a long time. Taking a lock and holding it for a duration of three seconds causes more than 1,000 other queries to queue up waiting for the lock to be released. In a massive system, namely a Grid middleware with hundreds of I/O operations per second, this can cause a significant impact on the performance of the system. A suboptimal temporary solution was implemented to recover from these situations by killing the database if more than 4,000 processes were queued up to access the database.

A user's priority is periodically recalculated based on a set of multiple parameters like resource usage and quotas set for the individual user. The legacy algorithm did not account for multi-core processing of jobs and explicitly handle each job as if it only uses a single core of CPU. Furthermore, the algorithm did not significantly penalize the use of large amounts of CPU-time, which can result in users with lower default priority being less likely to have their Grid jobs selected for execution. The legacy service optimizers trigger the algorithm to calculate the priority for users, track the resource usages and status of jobs.

An approach to this problem was to investigate the legacy algorithm, recreate it and improve fairness. Subsequently, reduce the locking time of the database to minimize the impact on the system. Moreover, the system would be improved by eliminating the issues where many processes are queued to access

the database. Furthermore, the optimizers in the previous service needed to be investigated, and their functionality recreated and enhanced in the new service to solve the database queue problems.

This project aims to determine how fairness can be improved in the user scheduling algorithm, and to reduce database locking times significantly.

1.3 Research questions

The research questions for this thesis are as follows:

1. **RQ 1**: What parameters are required in a user scheduling algorithm to ensure fairness and minimal impact on the performance of the system?
2. **RQ 2**: What is an optimized approach to sum and update user values, while minimizing system impact and having a short database lock duration?
3. **RQ 3**: How much can the total duration of the JAliEn TaskQueue database locking duration be reduced by splitting a large operation into smaller queries?

1.4 Contribution

The contributions from this work are primarily a novel user scheduling algorithm, and a set of services, called optimizers, that run on a specific interval. These contributions are deployed to the production environment of JAliEn, and are important parts of a system that can handle over a million batch jobs daily. The novel scheduling algorithm is used to calculate user priorities in a Grid middleware. This algorithm was tested through isolated simulations and tested by running in parallel with the legacy one, on the actual data from production, and comparing results. Furthermore, the optimizers are services that run on the central machines asynchronously to consolidate, synchronize and update information. The optimizers are responsible for tracking resource usage, job status, and user priorities. The optimizers are designed to reduce the database locking time and improve the performance of the system. Additionally, several flow state diagrams were created when developing the system to visualize the flow of the system.

The code for the algorithm and optimizers, and the state diagrams are available in the appendix A and D on pages 106 and 119.

1.5 Outline

The thesis is structured as follows:

Chapter 2
Provides a theoretical background on Grid computing, job scheduling, distributed computing, and database locking. This chapter also introduces the JAliEn Grid middleware, which is the case for this project.

Chapter 3
Presents the methodology used in this project, including the approach to solving the problem, the design of the new system, and the testing of the system.

Chapter 4
Presents the results of the project, including the novel user scheduling algorithm, the optimizers, and the performance of the system.

Chapter 5
Discusses the results and the implications of the project, including the limitations of the new system. The results of the research questions are discussed in this chapter.

Chapter 6
Concludes the thesis and summarizes the contributions of the project.

Chapter 7
Contains thoughts about the potential future work that could be done to improve the system further.

Chapter 2

Theory

The advent of computation can be compared, in terms of the breadth and depth of its impact on research and scholarship, to the invention of writing and the development of modern mathematics.

Ian Foster

This chapter provides the theoretical background necessary to understand the problem statement. It begins with an introduction to the scientific background of this field, followed by a brief exploration of the history of computing and the growing need for efficient utilization of computational resources. It then covers the concepts of job scheduling and distributed computing, with a focus on high-volume data processing and job management systems. Next, the Grid computing paradigm and the middleware that supports it are discussed. Additionally, an explanation of the JAliEn system—a real-world use case central to this thesis—will be presented. The chapter concludes by discussing the consistency and effectiveness of relational database management systems (RDBMS).

2.1 Job scheduling

In the early days of computing back in the 1950s and 1960s computers were large and expensive machines that primarily operated on batch processing systems. The resources were shared among multiple users who submitted their jobs to a queue to be processed sequentially. A major bottleneck in sequential computing is I/O operations which are significantly slower than CPU operations. This challenge led to the development of multiprogramming systems where activities are divided into a number of sequential processes that are placed in different levels in a hierarchy [6, 7]. These multiprogramming systems could hold multiple jobs in memory simultaneously to allow the CPU to switch to a different job when one had to wait for I/O. This approach was an early form of parallel computing where multiple processes are executed in parallel to increase the utilization of the CPU and resource sharing among users [8]. However, the increased CPU utilization added complexity to job scheduling, leading to a need for more sophisticated algorithms to optimize the utilization of system resources.

All modern operating systems use one or several variants of job scheduling algorithms to minimize idle time and maximize CPU utilization. In Linux, the Completely Fair Scheduler (CFS) is used by the kernel to schedule tasks, including user-space processes and kernel threads. It was recently replaced by the Earliest Eligible Virtual Deadline First (EEVDF) algorithm in version 6.6 of the Linux kernel [9, 10]. Scheduling is fundamental to the efficient operation of computer systems and is used in a wide range of applications, from personal computers to large-scale distributed systems. A process of execution consists of a cycle including two states, these are CPU execution and waiting for I/O. The duration of a CPU burst varies from process to process and computer to computer. According to Silberschatz et al. (2018), a process typically begins with a CPU burst, followed by an I/O burst, and this sequence repeats throughout its execution [11]. When the CPU becomes idle a new task must be selected for execution by the CPU scheduler that selects a process from memory and allocates the CPU to the process. The available tasks in memory are in a ready queue and these queues are not necessarily first-in, first-out, but can be implemented using a variety of scheduling algorithms. These algorithms include among others First-Come, First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), Priority Scheduling and Multilevel Queue Scheduling [11]. Each of these scheduling

algorithms will be explored in more detail later in this section.

The following presentation of established principles for job scheduling algorithms is based on Silberschatz et al. (2018) [11], there are two relevant modes of scheduling called **non-preemptive** and **preemptive** scheduling. In non-preemptive scheduling a process allocated to the CPU will be kept by the CPU until it is released by termination or by switching to the waiting state. In preemptive scheduling the CPU can be taken away from a process and allocated to another process. Preemptive scheduling is more common in modern operating systems as it allows for better response times and fairness among users [11]. CPU-scheduling decisions are made in several situations, number 1 and 4 in the enumerated list below are non-preemptive, while 2 and 3 are preemptive [11]:

1. When a running process needs to wait, such as for an I/O operation.
2. When a running process is preempted and moves back to the ready state, for example, due to an interrupt.
3. When a waiting process becomes ready again, for example after an I/O operation completes.
4. When a process completes its execution and terminates.

The scheduler not only selects the next process for the CPU, but also depends on a component called the dispatcher to manage key tasks. These tasks include switching context between processes, resuming programs and switching to user mode. The dispatcher will be invoked on every context switch, therefore it should be as fast as possible with minimal latency when stopping and starting processes [11]. Furthermore, there are several criteria that can be used to evaluate the performance of a scheduling algorithm, these include **CPU utilization**, **Throughput**, **Turnaround time**, **Waiting time** and **Response time** [11]. The first criterion concerns CPU utilization, which is measured as the percentage of time the CPU is busy. Throughput is the number of processes completed per time unit. Turnaround time is the total time spent waiting and executing a process. The second to last criteria is the waiting time, this is defined as the sum of time which a process spends waiting in the ready queue. Finally, the response time is the time from when a request was submitted until the first response is produced. In computer systems it is desirable to maximize the CPU utilization and throughput while minimizing the turnaround time, waiting time and response time. The sig-

nificance of different scheduling criteria can vary based on the specific system and use case. For example, in a personal computer, achieving a predictable response time is typically prioritized over a variable response time [11].

First come, first served

First come, first served (FCFS) is a simple and traditional scheduling algorithm that is managed with a First In, First Out (FIFO) queue. The first process that arrives in the queue is the first to be executed. As this scheduling algorithm is non-preemptive it won't be interrupted by other processes and will run until execution is completed and it terminates, or it needs to wait for an I/O operation. The algorithm is easy to implement, however the average wait time is longer than other scheduling algorithms. With larger processes that arrive first, there is a convoy effect as smaller processes are forced to wait for the larger processes to complete. The result of this convoy effect is that the average wait time increases [11]. Figure 2.1 below is a recreation of a Gantt chart illustrating the FCFS scheduling algorithm from Operating System Concepts by Silberschatz et al.

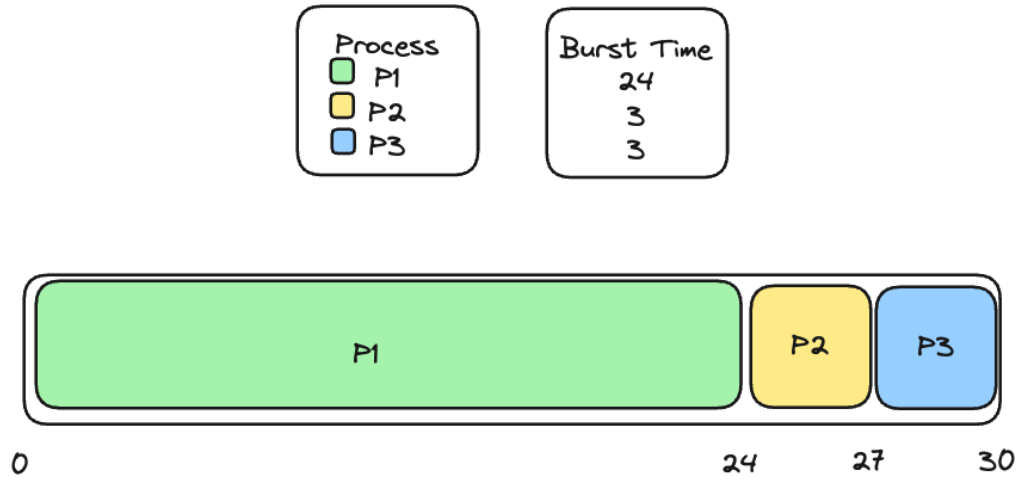

Figure 2.1: A recreation of a Gantt chart illustrating the **First Come, First Served** scheduling algorithm from Operating System Concepts by Silberschatz et al. [11].

Shortest Job First

Shortest Job First (SJF) is another scheduling algorithm that selects the process with the smallest execution time (CPU burst) next. If two or more processes have the same execution time FCFS will be used to determine the order. The SJF algorithm determines the next task based on the next CPU burst, rather than the total length of the entire task. However, at the CPU level, this algorithm is not feasible to implement because the length of the next CPU burst is unknown. Instead, it is possible to predict the next burst length based on previous bursts, allowing the scheduler to approximate which process will have the shortest predicted CPU burst. The algorithm can be implemented in a non-preemptive or preemptive mode, both of which are illustrated in Figures 2.2 and 2.3. In preemptive mode, the algorithm is often called Shortest Remaining Time First (SRTF). Upon arrival of a new process, the scheduler compares its CPU burst time with the remaining burst time of the current process. If the new process has a shorter burst time, it will preempt the current process. In non-preemptive mode, the process will run until it completes it's CPU burst [11]. Figures 2.2 and 2.3 below is a recreation of a Gantt chart illustrating the SJF scheduling algorithm as non-preemptive and preemptive, from Operating System Concepts by Silberschatz et al.

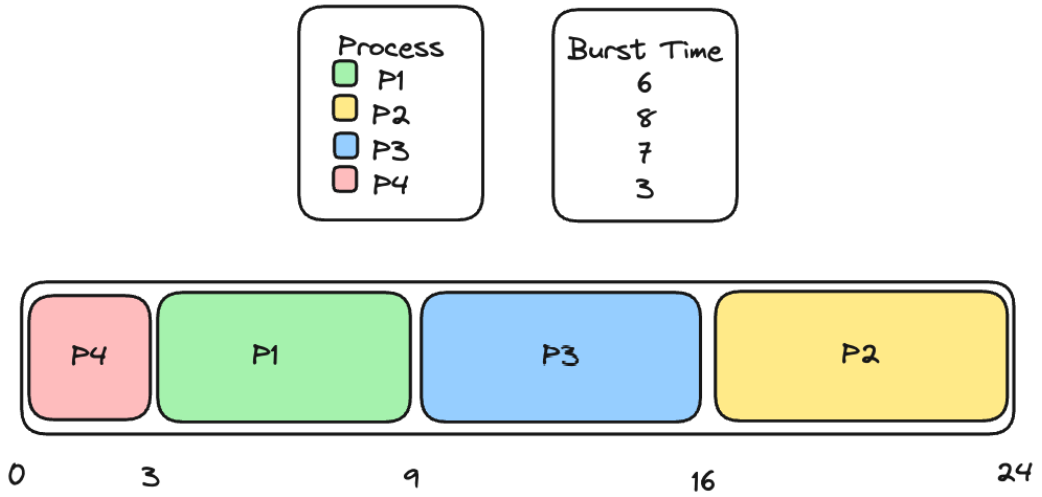

Figure 2.2: A recreation of a Gantt chart illustrating the non-preemptive mode of the **Shortest Job First** scheduling algorithm from Operating System Concepts by Silberschatz et al. [11].

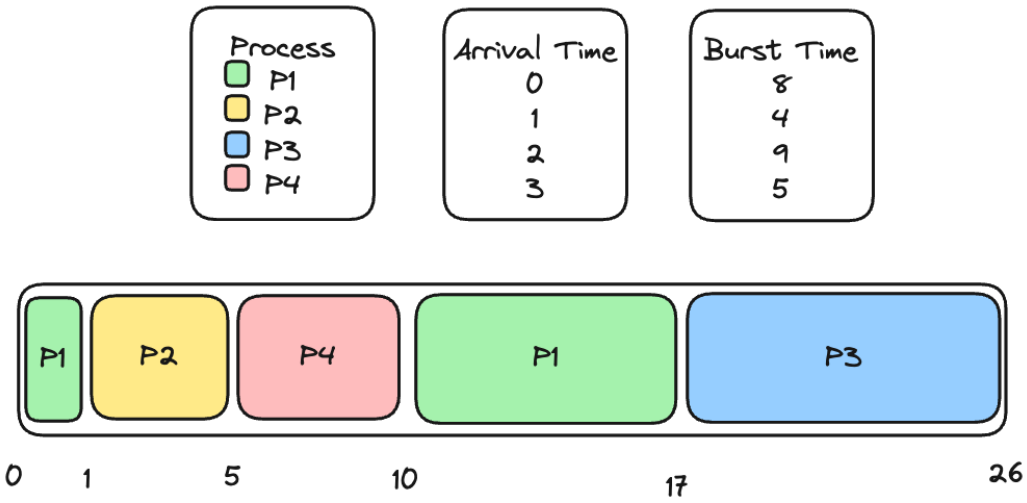

Figure 2.3: A recreation of a Gantt chart illustrating the preemptive mode of the **Shortest Job First** scheduling algorithm from Operating System Concepts by Silberschatz et al. [11].

Round Robin

Round Robin (RR) is a scheduling algorithm that incorporates preemption to let the system switch between processes. The algorithm is designed to allocate a short unit of time called a time quantum or time slice to each process in the queue. The CPU scheduler will cycle through the items in the ready queue and allocate one time slice to each process to be executed on the CPU. If the process has not completed execution after one time slice it will be put to the back of the queue and receive another time slice when the scheduler cycles through the queue again. If a process completes in less than one time slice the process will be released and the next one will be allocated from the queue. The time quantum is typically between 10 and 100 milliseconds and is determined by the system [11]. Optimizing the performance of the Round Robin algorithm requires selecting an appropriate time slice—not too short to avoid excessive context switching, nor too long to maintain system responsiveness. Silberschatz et al. suggest using about 80% of the average CPU burst time as a practical guideline. Figure 2.4 below is a recreation of a Gantt chart illustrating the RR scheduling algorithm from Operating System Concepts by Silberschatz et al. This figure uses a time slice of 4 milliseconds.

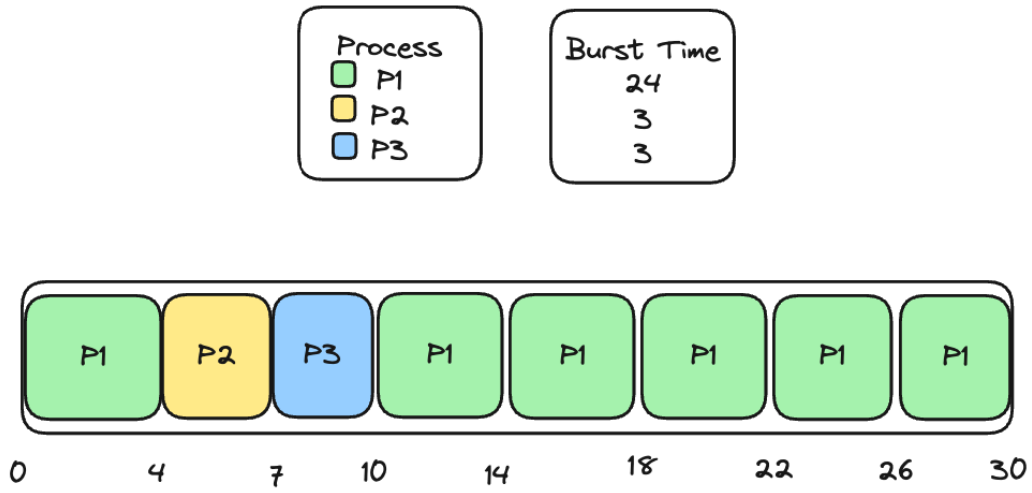

Figure 2.4: A recreation of a Gantt chart illustrating the **Round Robin** scheduling algorithm with a 4 ms time slice, from Operating System Concepts by Silberschatz et al. [11].

Priority Scheduling

Priority Scheduling (PS) is a algorithm where a priority is associated with each process and the process with the highest priority is allocated to the CPU. If two processes have equal priority, then FCFS is used to determine the order. The SJF algorithm is essentially a variation of the Priority Scheduling algorithm, where priority is determined as the inverse of the predicted next CPU burst. Consequently, processes with shorter predicted CPU bursts are assigned higher priority. The priorities are defined internally or externally. Examples of internal priorities are memory requirements, time limits or I/O requirements. External priorities are criteria set outside of the system such as the importance of the process or the submitting entity [11]. Figure 2.5 is a Gantt chart illustrating the PS algorithm in a non-preemptive mode. Priority Scheduling can operate in either preemptive or non-preemptive mode. As described in the list on page 20, the Priority Scheduling algorithm in preemptive mode will preempt the CPU if a newly arriving process has a higher priority. Furthermore, there is a significant problem of starvation or indefinite blocking that could occur for processes with low priority. A solution to this is to implement aging, where the priority of the process would gradually increase as it waits in queue. Another solution is to combine Round Robin

and Priority, allowing the system to execute processes based on priority, while using a Round Robin approach to manage processes with equal priority levels. This approach can be described as a Multilevel Queue Scheduling algorithm and will be introduced in the next paragraph [11].

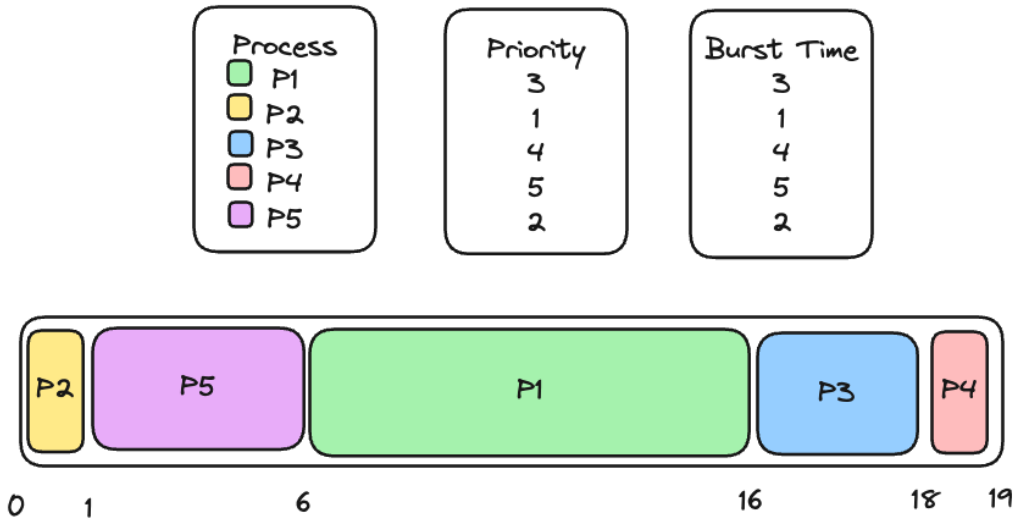

Figure 2.5: A recreation of a Gantt chart illustrating the **Priority Scheduling** algorithm from Operating System Concepts by Silberschatz et al. [11]. The processes are assigned a priority of 1 to 5, with 1 being the highest priority.

Multilevel Queue Scheduling

A **Multilevel Queue Scheduling (MQS)** consists of multiple queues with different priority levels where each level can have a distinct scheduling algorithm. Additionally, scheduling must also be performed among the queues, which is often implemented using preemptive scheduling with fixed priorities assigned to each queue. The processes in the highest priority queue are scheduled first. The multilevel queues can be used to separate processes by type, a typical separation is interactive (foreground) and batch (background) processes. These might have different requirements to response time and priority. For example can Priority Scheduling be combined with Round Robin to execute processes in the highest prioritized queue in RR order [11]. Figure 2.6 is an illustration of priority levels in a MQS algorithm from highest to

lowest priority, recreated from Operating System Concepts by Silberschatz et al. Any process in a lower priority queue can only be executed when the higher priority queues are empty. However, it is possible to use time slices between the queues to allow each queue a specific portion of the CPU time using its own scheduling algorithm [11]. The MQS approach is prone to starvation of the processes in the lower queues. Although it has low overhead, it is inflexible, as processes cannot move between queues. A solution to prevent starvation is to implement aging, which is achieved through the Multilevel Feedback Queue (MFQ) Scheduling algorithm, allowing processes to move between queues based on the characteristics of their CPU bursts. The MFQ algorithm has a set number of queues and a specific scheduling algorithm for each queue. Additionally, it has a mechanism to determine when a process should be promoted or demoted to a different queue. The Multilevel Feedback Queue algorithm prioritizes processes with shorter CPU bursts, while those with longer bursts receive lower priority [11].

The above introduction to several well-known scheduling algorithms aims to provide an overview of the need for job scheduling, along with its different concepts and approaches. Although these scheduling algorithms are primarily used in operating systems, the concepts are relatable to scheduling challenges in distributed computing systems. For this project there are concepts from Priority Scheduling and Multilevel Queue Scheduling that are relevant to develop an algorithm to calculate the priority of a user. More on this in Section 3.4.2 (p. 57). The next section will introduce the concept of high volume data processing in distributed systems and the need for efficient job scheduling and job management in large-scale distributed systems.

2.2 High volume data processing with distributed computing

Distributed computing involves utilizing multiple computers, which may be located in different geographical areas, to solve complex computational tasks collaboratively. These computers are connected through a wide-area network and can communicate either with each other or with a central service, depending on the use case. The main goal of distributed computing is to solve problems that are too large for a single computer to handle, often involving the processing of high volumes of data. This can involve solving a

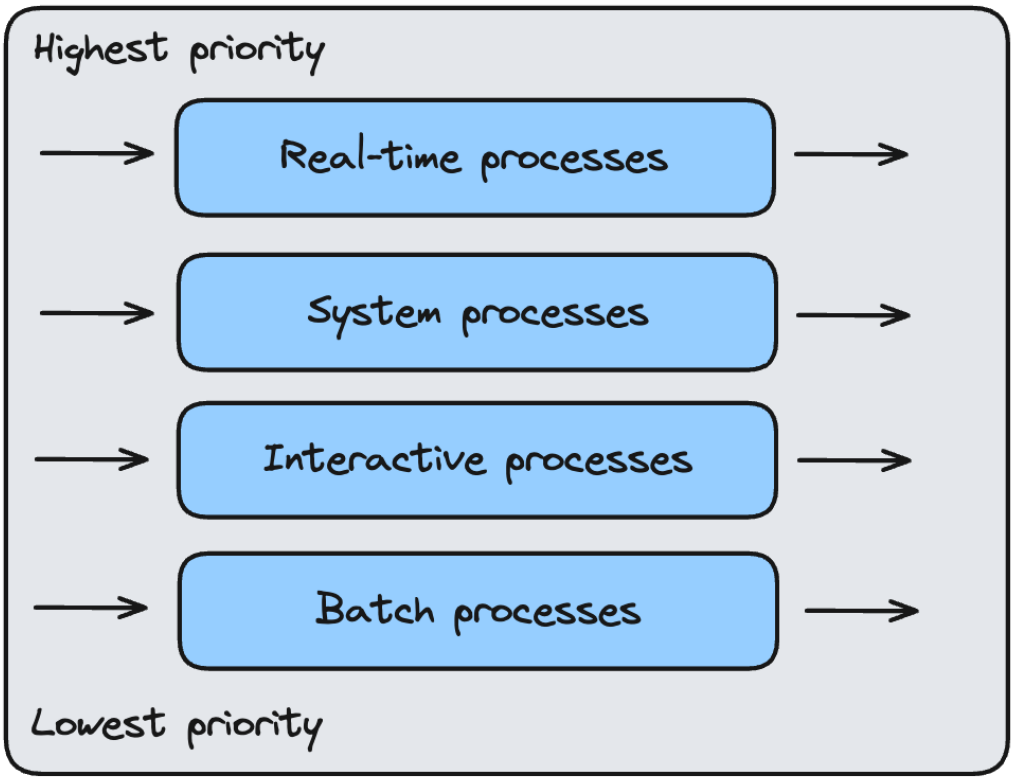

Figure 2.6: A recreation of an illustration of priority levels in a **Multilevel Queue Scheduling** algorithm from Operating System Concepts by Silberschatz et al. [11].

vast number of tasks or a single task that requires immense computational power. There are several variants of distributed computing with different goals. **High performance computing** (HPC) is a type of distributed computing that focuses on solving complex problems in the shortest amount of time. HPC tasks typically focus on CPU-intensive computations, with workloads characterized by metrics such as scalability and performance, measured in terms of floating-point operations per second (FLOPS) [12]. A workload often consists of a single task executed as a single job. Supercomputers are typically designed for high performance and are designed for HPC workloads. **High throughput computing** (HTC) is a type of distributed computing that focuses on maximizing the volume of work completed, typically by splitting larger tasks into smaller, independent jobs. These tasks can commonly

be large-scale data analysis from scientific experiments, simulations or long-running batch processing workloads [12]. Large scale experiments such as the ALICE and ATLAS experiments at CERN utilize HTC workloads and each have access to over 200,000 CPU cores and hundreds of petabytes of storage through the WLCG [3, 4, 12]. To efficiently manage distributed HTC workloads and resources, job scheduling and job management systems are essential.

2.2.1 Job management systems

Job management systems are essential in high volume data processing to allocate and manage the executions of jobs and for efficient utilization of resources in distributed computing systems. These job management systems optimize the management of tasks and enable organizations and researchers to maximize the performance of available computational resources. These may be located locally or distributed across multiple sites and connected by a network. The system must be able to handle a large number of jobs and efficiently allocate resources to minimize idle time. An example from JAliEn monitoring in Alimonitor [13] shows that on the ALICE Grid, with 200,000 CPU cores, there can be up to 1.5 million jobs, necessitating approximately 17 scheduling decisions per second for new jobs. There are multiple job management systems and schedulers designed for distributed computing environments, with some tailored to High Throughput Computing (HTC) workloads and others to High Performance Computing (HPC) tasks. Prominent examples of these systems include HTCondor, SLURM, TORQUE, Portable Batch System (PBS) and Load Sharing Facility (LSF) [14]. Furthermore, the container orchestration tool Kubernetes can be leveraged to manage resources that are available for the job management systems [15]. The focus of the next paragraphs will be on HTCondor and SLURM as they are widely used in distributed computing environments and are relevant in JAliEn.

HTCondor is a batch workload management system where the primary goal is to utilize all available batch slots as quickly as possible. Previously known as Condor, the batch system is mainly utilized in high throughput computing. The batch system can aggregate resources from multiple resource providers to be applied in large-scale distributed computing, including Grid computing [15, 16]. HTCondor holds a pool of available batch slots to execute jobs and will try to fill these as quickly as possible to leverage computational resources

on a local or remote site. These batch slots can be distributed crossing physical and organizational boundaries [15]. Utilizing these resources over time is the primary purpose and consideration of fairness is secondary, but balanced over time. The batch jobs to be executed are submitted in the form of Job Description files that define the requirements of the execution environment, such as available disk space, memory or number of cores. HTCondor supports the execution of multi-node jobs, jobs that need to intercommunication, by utilizing the Message Passing Interface (MPI). However, this requires that the jobs are submitted from a single host using a dedicated scheduler. In HTCondor this scheduler is called *condor_schedd* [16]. HTCondor supports preemption and can preempt batch jobs to reschedule them on a different computational resource [15]. However, not all batch queues support suspending and resuming jobs, this functionality is not available in JAliEn.

Like HTCondor, SLURM is an open-source batch system for HPC and HTC use cases. It is especially popular in HPC and many of the most powerful supercomputers in the world uses SLURM [17, 18]. A multitude of functionality is supported, like sharing of resources across multi-node environments, scheduling Graphics Processing Unit (GPU) resources, resource partitioning and node scheduling. Moreover, SLURM is able to make scheduling decisions based on network-topology [16]. The batch system is highly scalable and manages job scheduling across Linux clusters of varying sizes. SLURM has three primary functions, the first is to allocate exclusive and/or non-exclusive access to computing nodes for a duration of time. The second is to provide a framework for monitoring, executing and starting jobs on the allocated nodes. Finally, the third function is to manage the resources and the pending work to be handled through a queue. SLURM utilizes a central manager and a set of daemons to manage jobs and resources. User commands include the ability to initiate jobs, monitor job information and their progress, in addition to terminating a running or queued job [17]. Both of the job management systems introduced above are widely used in distributed computing and both are supported and used in JAliEn.

2.2.2 Grid computing

The concept of Grid computing is *coordinating resource sharing and problem-solving in dynamic, multi-institutional virtual organizations*, as stated by

Foster et al. in *The Anatomy of the Grid* [19]. The term Grid was proposed in the late 1990s to describe a branch of distributed computing focused on large-scale resource sharing across wide-area networks. The need for Grid computing arises from large-scale collaborations that aim to solve complex problems requiring immense computational resources. Key features of the Grid include collaborative problem-solving and resource-brokering strategies to efficiently manage and allocate resources. The shared resources, such as storage, CPUs and GPUs are distributed across multiple physical locations and connected by wide-area networks. Files and software can also be shared and utilized by virtual organizations (VOs) on the Grid [19, 20].

The sharing of resources on the Grid must be coordinated through establishing and enforcing sharing agreements. A set of institutions and individuals that adhere to these agreements form a *Virtual Organization* (VO) [20]. The participants of a VO can dynamically share resource across heterogeneous platforms, languages and programming environments. A VO is a complement to, not a substitute for existing institutions. For Virtual Organizations to share resources effectively, they must agree on a common set of Grid protocols to ensure interoperability. These Grid protocols are commonly implemented in the form of Application Programming Interfaces (APIs) or Software Development Kits (SDKs) to provide multilevel programming interfaces [20]. A multilevel programming interface provides different layers of abstraction for interacting with software components. In the context of APIs and SDKs this means that access is offered on various levels such as high-level user commands and lower-level functions available to the developer [20]. Together this architecture and technology constitute a middleware. A middleware is *the services needed to support a common set of applications in a distributed network environment* [19, 21]. Grid middleware will be discussed further in Section 2.2.3. The Grid protocols are layered on top of the Internet protocols and are presented in Figure 2.7 and explained in the next paragraph.

The definition of a protocol specifies the structure of the information to be exchanged and how the elements of a distributed system interact to achieve a desired behavior [19]. The top layer of the Grid protocol architecture is the **application** layer that incorporates the user applications that runs within a VO environment. By adhering to defined protocols in the VO environment and utilizing APIs, the application layer promotes code reusability and portability. The next layer is the **collective** layer that in addition to APIs

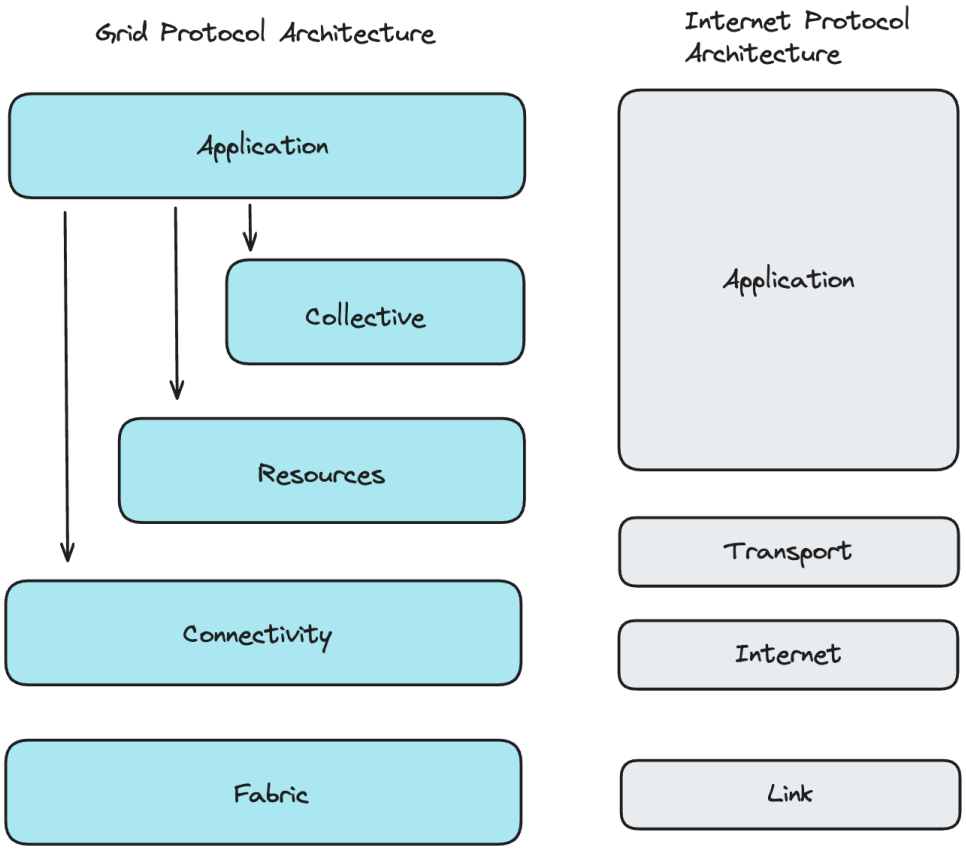

Figure 2.7: The layered interaction between the Grid architecture and the Internet protocol architecture - reproduced from [19].

and SDKs contain the services and protocols that are global and capture interactions over a collection of resources [19]. Components of this layer builds upon the connectivity and resources layers that implement multiple behaviours. Examples of these behaviors include, but are not limited to, monitoring and diagnostic services, directory services such as discovery of VO resources and Grid-enabled programming systems like the Message Passing Interface [22, 23]. The **resource** layer builds upon protocols from the connectivity layer, specifically connectivity and authentication. Contrary to the collective layer where the global state is considered, the resource layer is concerned with individual resources. This layer contains protocols to access

information about the state and structure of a resource. Furthermore, it has management protocols that are responsible for instantiating resource sharing connections [19]. Services in the authentication protocols of the **connectivity** layer provide cryptographic mechanisms to verify user and resource identities. Additionally, the connectivity layer contains the requirements for communication which include routing, transport and naming. In most situations these communication protocols utilize the TCP/IP stack, specifically the internet, application and transport layers of the Internet protocol architecture, see Figure 2.7 [19, 24]. The components of the **fabric** layer implement local and resource-specific operations that interact with physical or logical resources. These resources encompass computational, storage, and network resources, as well as code repositories and catalogs. The computational, storage and network resources all have inquiry functions that resolve the software and hardware characteristics of the resources. This information also include relevant load information, such as available storage space and CPU load. All of these Grid protocols are explained in detail in the paper by Foster et al. [19].

The paper by Foster et al. also discusses the Globus Toolkit, a set of software tools that primarily operates at the fabric layer, providing services required to build a Grid environment. The Globus Toolkit (GT) provides inquiry software to discover state and structure information from a variety of resources, such as computers, storage systems and network. Additionally, the GT provides a small set of standardized protocols for the resource layer and security protocols for the connectivity layer. The protocols are the following:

- **GSI** - The Grid Security Infrastructure, which provides a secure communication channel between Grid services and clients. The GSI is public key-based and uses X.509 certificates [25]. GSI protocols are used for authentication, authorization and communication. Additionally, it builds upon the Transport Layer Security (TLS) protocols [26] that is utilized for single-sign on, delegation, user-based trust and integration with existing security systems.
- **LDAP** - The Lightweight Directory Access Protocol [27], a protocol used to access and manage directory information. Additionally, this protocol is used to define a standard information protocol. This includes the associated Grid Resource Registration Protocol that is used to register resources with the Grid Index Information Servers [19].

- **GridFTP** - An extended version of the File Transfer Protocol (FTP) that is utilized for data access and provides reliable and secure data transfer between Grid sites [28].
- **GRAM** - The Grid Resource and Access Management protocol is based on HTTP and is utilized to allocate and monitor computational resources and to manage computation on the resources.

At the onset of the Grid computing era, the Globus Toolkit was virtually synonymous with Grid computing. Though now considered outdated and no longer supported, the Globus Toolkit's legacy continues through Globus Connect, a modern cloud service adopted by major companies like Microsoft and Google [29]. The concepts and protocols from the GT are still relevant and more modern solutions build upon these concepts. For example, the need to move large files across the Grid is still relevant, but the GridFTP protocol has been supplemented by more modern solutions like XRootD and Fast Data Transfer (FDT). The XRootD protocol is primarily designed for distributed file access and data transfers within large-scale computing environments, while FDT (Fast Data Transfer) is a Java-based application optimized for high-speed data transfers across wide-area networks and large geographical distances. [30, 31].

Fast Data Transfer (FDT) is an application designed for efficient data transfers over wide-area networks, with capabilities for reading and writing data to disks [30]. Written in Java, it utilizes the Java NIO (New Input/Output) libraries and runs on the JVM. The application is built to be an asynchronous and multithreaded system and the main features are the following: continuous buffered streaming of data, parallel data transfers, recovery from network failures and the ability to read and write to disk while utilizing multiple streams simultaneously. Any file transfer that might fail can be retried without loss [30, 32].

XRootD is an extended version of *rootd* that allows a single server to act as a uniform interface for data access from multiple load balanced environments. This interface is defined as a communication protocol that define the functionalities and interactions available to the clients [31]. The functionality of the interface includes the following: inquiry for information about resource location, authentication and authorization to XRootD systems and data access. Additionally, XRootD include retry capabilities and will try to identify a different server or retry the same server if a connection fails. A

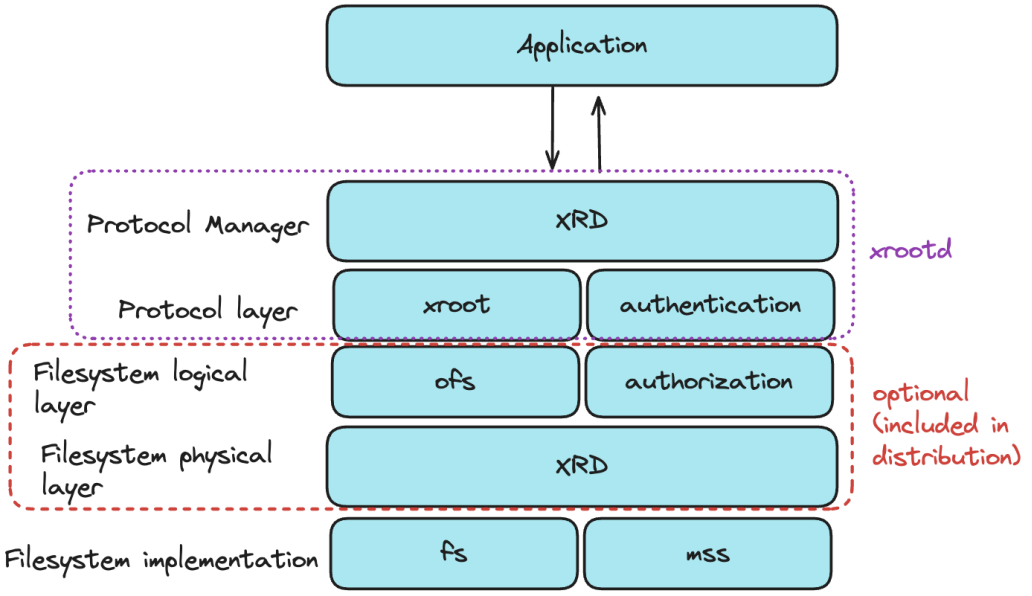

Figure 2.8: The layered architecture of XRootD displaying mandatory and optional layers that can be combined as required. Recreated from [31].

single TCP connection from a client to a server can be used to open multiple data streams, which may also be accessed by other clients [30, 31]. The layers of the XRootD architecture are shown in Figure 2.8. Notably, each layer is sufficiently isolated, allowing them to be dynamically loaded, which enables support for a variety of different implementations [31].

The potential use cases for virtual organizations and the Grid are many, ranging from scientific research to business collaborations and crisis management [19]. A concrete case is the ALICE experiment at CERN, where the ALICE VO is a virtual organization that collaborates on analyzing physics data from the Large Hadron Collider (LHC). To analyze the petabytes of data collected in the experiment it is necessary to pool together storage, networking and computational resources to create a Data Grid. A Data Grid is a Grid designed for sharing and management of distributed datasets [33]. Figure 2.9 (p. 38) displays the entire VO and includes the grid sites with worker nodes, central services and the user machines.

2.2.3 Grid middleware

A Grid middleware is a software layer between the application layer and the hardware infrastructure that provides the essential capabilities needed to connect to and leverage a distributed computing environment. This enables resource sharing over a multi-institutional virtual organization to solve complex problems [19, 34]. The Grid middleware acts as an intermediary layer between physical resources (like computing nodes, storage and networks) and the software applications required to utilize those resources. The essential capabilities of a Grid middleware includes job scheduling, authentication, data transfer and monitoring. Job scheduling is a fundamental function of Grid middleware, responsible for allocating resources, managing workflows, and scheduling jobs for execution. A Grid middleware must also manage resources across the Grid and provide security services for the grid applications, including authentication, authorization and encryption. To execute jobs effectively on multiple physical sites, the middleware must manage data transfers between the nodes and be informed about the location of data and resources to make informed decisions on where to execute jobs. The middleware must provide monitoring services to track the status of jobs, resources, and data transfers [34]. Moreover, a variety of Grid middleware solutions are available, each offering a unique set of features and functionalities in addition to the essential capabilities [19]. Historically, some of the most widely used Grid middlewares included the Globus Toolkit, gLite, and AliEn, which are no longer in use, alongside ARC, Globus Connect and UNICORE, which continue to be utilized [29, 35]. Moreover, JAliEn is the modern version of the AliEn middleware and is currently in use by the ALICE VO at CERN to access the computational and storage resources available on the WLCG. The following section will examine the JAliEn middleware, the case study for this thesis, in greater detail.

2.2.4 Case: JAliEn

The Grid middleware JAliEn provides a unified interface for users to access the files, submit batch jobs and monitor jobs on the WLCG. JAliEn is a heterogeneous system where the underlying architecture and operating system on the computational nodes vary, this heterogeneous complexity is unified by JAliEn to look like a single system for the user. The JAliEn system consists of multiple services, they can be divided into central services (JCentral), site

services, and an authentication service (JBox) [3]. Users may submit batch jobs to JAliEn by submitting Job Description Language (JDL) files to request certain parameters such as memory, disk and CPU requirements for their batch jobs. Users can define a parameter in the JDL to instruct JAliEn to split the work into independent pieces based on the specified criteria in the JDL and data locality. Work that has been split into independent pieces is referred to as a master job and its subjobs. A master job must have one or more subjobs, and each subjob is associated with a single master job.

JCentral is the central service that is responsible for direct database connections, assigning job execution site and resolving API-requests [36]. Furthermore, the JCentral is responsible for data management and will make decisions on where files for grid jobs should be stored. The JCentral handles all functions related to the job queue (TaskQueue), such as batch job submission, status, splitting and management [3]. Moreover, the JCentral manages the quotas for user and the recalculation of user priorities. The service also handles information about resource usage and the status of jobs. An architecture overview of the JAliEn system is shown in Figure 2.9.

A VOBox is an experiment-specific service machine that interfaces the site resources to the central experiment services. In the case of ALICE it runs two services: the Computing Element and MonALISA [3, 13]. The JAliEn Computing Element service injects generic jobs in the site local batch queue, as instructed by the JCentral components (when there is work to be done that matches the site). The generic (pilot) jobs start a JobAgent JAliEn component that advertises the node resources and retrieves user submitted (grid) jobs from the central queue. During execution, all JAliEn components report monitoring data to the MonALISA service running on the VOBox, allowing for example real time inspecting of job behavior or storage network traffic, as well as aggregating resource usage for accounting purposes. These resources include memory, disk and CPU. The JobAgent requests a matching grid job from JCentral, and upon receiving a job, the job payloads are executed on the site. During execution, the JobAgent regularly reports the job's status to JCentral. Upon completion of a job, the resource usage and the job's final status are reported to JCentral. This resource information is relevant to determine the status of a user's quota and to calculate priority for users [3, 13]. The maximum execution time for a task is specified by the user as a parameter in the JDL when submitting a job.

For all the worker nodes the software is distributed through the CernVM File System (CVMFS). CVMFS is a read-only file system specifically designed to support high-energy physics experiments, such as ALICE. It provides a reliable, fast and scalable method for distributing analysis software to grid worker nodes and virtual machines [37].

Furthermore, the system is secured by utilizing the X.509 Public Key Infrastructure (PKI) [40] that ensures that only authorized users can access the grid resources. JAliEn extends upon this functionality and has its own Certificate Authority(CA) which is used to obtain signed Token Certificates (TC) for services within the system [3]. The Token Certificate is a signed time-limited certificate that represents a job, user or a pilot submitter [41]. These TCs grant the ability to differentiate token information and identify the requesting service, such as the JobAgent or user payload. Moreover, every request is authorized by the JCentral [3]. Users can connect to JAliEn by utilizing a Grid certificate and to get a Linux like console shell [3]. On the end-user machine the daemon service JBox is running to handle authentication upstream using the user's X.509 grid certificate [36].

Given the size and significance of the ALICE experiment, a major research project at CERN, the JAliEn system is a critical component in data processing. This system employs job scheduling to manage resources across numerous sites and users. The system's heterogeneity and the vast amount of data to be processed make it an excellent use case for a project aimed at optimizing job scheduling in a real system. This optimization is crucial for improving the Grid middleware components that are essential for investigating the fundamental building blocks of our universe.

The JAliEn database

In JAliEn there are several central databases, the three prominent ones are called TaskQueue, Catalogue and Transfer. These are displayed on the JAliEn architecture overview in Figure 2.9. The focus in this thesis is on the TaskQueue database which stores job-related data, including information about users, jobs, resource usage, and site details. The database is a MySQL database that is accessed by the JCentral service. At the time of writing the TaskQueue database has fifteen tables, some of the most important tables, and the focus for this thesis, are the following: SITEQUEUES, JOBAGENT, PRIORITY, QUEUE, and QUEUEPROC. A brief description

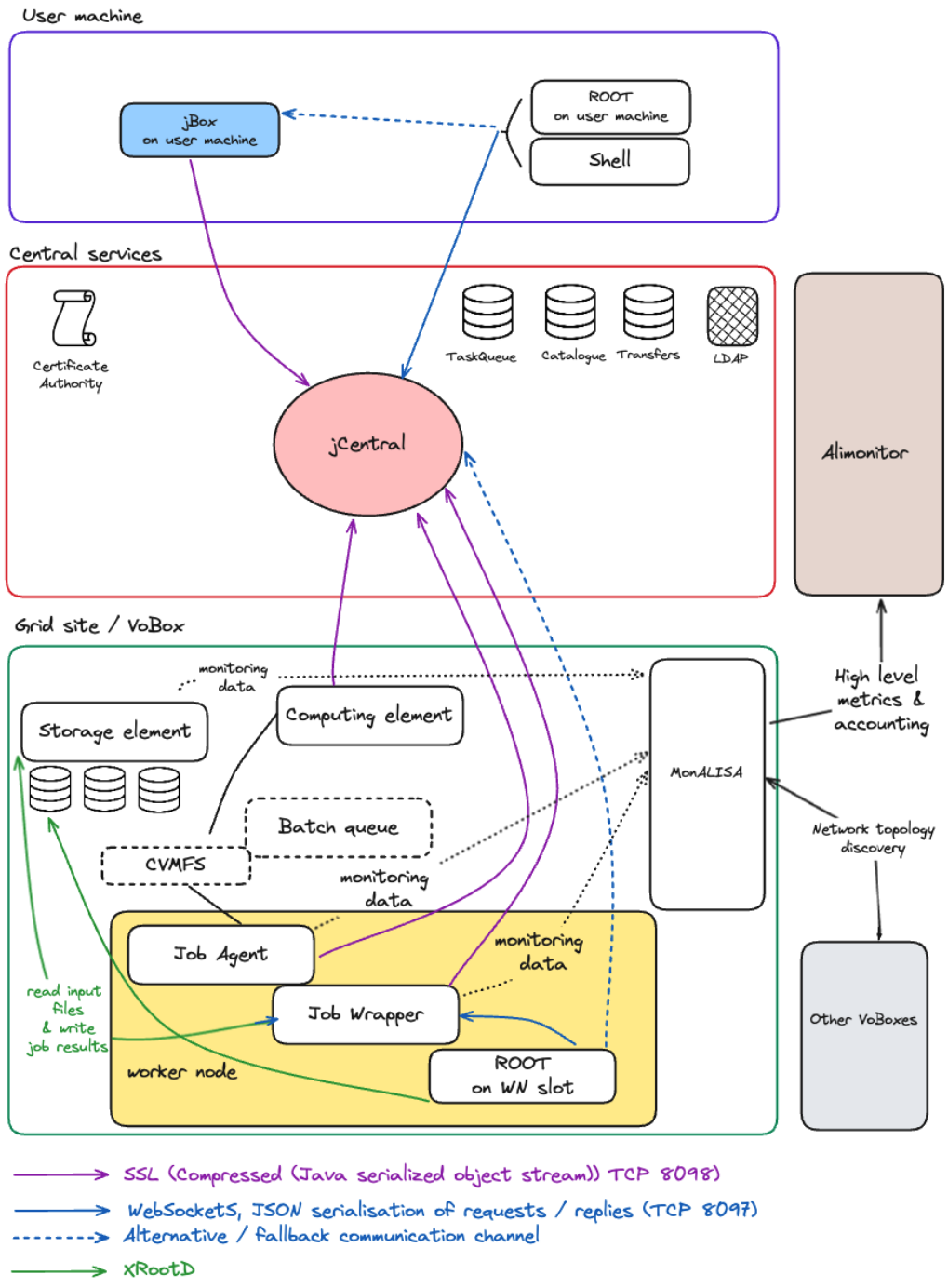

Figure 2.9: ALICE VO components used in JAliEn, based on [5, p. 2], [38, pp. 20–21] and [39, pp. 5–6].

of these tables is given below, the table columns are listed in Appendix C.

- SITEQUEUES: This table stores the status of each site, detailing the number of jobs in various states. See C.1 on page 115 for a detailed list of columns.
- JOBAGENT: This table is used by job agents to find potential matches for their nodes. Each entry corresponds to jobs in the WAITING state and they have an associated *userId*, with the *priority* reflecting the *computedPriority* from the PRIORITY table. Moreover, the JOBAGENT table contains information about the constraints that should be used for matching. See C.2 on page 115 for a detailed list of columns.
- PRIORITY: This table contains the user's quotas and priority within the system. The priority is determined based on various factors including baseline priority, resource consumption, and quotas. See C.3 on page 116 for a detailed list of columns.
- QUEUE: This table tracks the jobs within the system, including their status, the user who submitted them, and the site where they are running. It also includes details about both master and subjobs. Larger jobs might be split into multiple smaller jobs, here a job will be assigned as the master and the rest will be subjobs. A master job has its own ID, but it is not a job to be executed, instead each subjob entry in the table has a field know as *split*, which equals the ID of the master job. See C.4 on page 117 for a detailed list of columns.
- QUEUEPROC: This table focuses on jobs running on the sites, detailing their status, resource usage, and the user who submitted them. The data from this table is crucial for synchronizing user resource usage and is used to calculate the user's priority. Resource usage information is persisted in this table when it is reported from the computing nodes. See C.5 on page 118 for a detailed list of columns.

2.3 Consistency and effectiveness in RDBMS

The focus for this section is the consistency and effectiveness in relational database management systems (RDBMS), such as MySQL and PostgreSQL. Furthermore, other concepts such as optimistic concurrency control, eventual consistency, the CAP theorem and reading committed or uncommitted data will also be considered. No forms of NoSQL databases are considered here, as they are not relevant to the case discussed in this thesis.

Relational database management systems, like MySQL and PostgreSQL, are structured to store information in tables composed of rows and columns. Each row represents a record, while each column corresponds to a specific field within that record. Tables in relational databases use primary keys to uniquely identify each record, and foreign keys to establish relationships between tables. Relationships enforced by the database server are for example one-to-one, one-to-many and many-to-one. This relational design enables the use of referential integrity which is a technique utilized to maintain data in a consistent format [42]. Referential integrity is a part of the ACID philosophy, which stands for Atomicity, Consistency, Isolation, and Durability. Through relational integrity data is kept consistent by using foreign key constraints to automatically update changes to related tables or prevent changes that would violate the integrity of the data [42, 43].

The ACID acronym represents a set of desirable properties in a database system, closely related to transaction processing [44]. A transaction is an **atomic** unit of work that adheres to the ACID principles: a transaction can be committed, fully completed, or rolled back, not completed. Meaning that when changes are made to the database in a transaction, they either all succeed or all fail [45]. The **consistency** property ensures that the database remains in a consistent state at all times, before, after and during transactions. The **isolation** property ensures that transactions are protected from another while in progress, meaning that they cannot interfere with each other or read uncommitted data. The isolation of transactions is obtained through the locking mechanism by setting the isolation level [44]. MySQL and MariaDB, which use the InnoDB storage engine [46], support four isolation levels as defined by the SQL:1992 standard: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE [47]. The default isolation level for InnoDB is REPEATABLE READ which enforces a high degree of consistency and allows multiple non-locking SELECT state-

ments within the same transaction. Within the same transactions the read will use the snapshot established by the first read when using the default isolation level. Using READ COMMITTED as isolation level will refresh and read the new snapshot during a transaction. Allowing transactions to read uncommitted data with isolation level READ UNCOMMITTED, known as a dirty read, can yield performance and concurrency gains. This does not ensure the consistency of the data read by all users, but does not lock the data in the row for reading. The final isolation level is SERIALIZABLE, InnoDB prevents interference between transactions by converting plain SELECT statements to SELECT ... FOR SHARE when auto commit is disabled. This level maintains the strictest consistency, avoiding phantom reads and ensuring isolation [44, 47]. In real-world scenarios, a compromise between the perfect transaction isolation and performance must usually be considered [48]. Reading uncommitted data does involve a possibility of a earlier version of the row being read instead of the latest version [47]. The final property of ACID, **durability**, ensures that once a transaction is committed, its changes are permanent and persisted even in the event of a system failure [44].

In addition to the topics covered above, several other relevant RDBMS features—such as concurrency, optimistic strategies and the CAP theorem are pertinent to this thesis and will be briefly introduced here. Concurrency in database transactions is the ability to run multiple operations simultaneously and not interfer with other operations [49]. This protection to allow concurrent operations has a low performance overhead and is achieved by using efficient locking mechanisms. A commonly used method for non-locking concurrency control is Multiversion Concurrency Control, which InnoDB employs to achieve consistent reads [50, 51]. There are optimistic and pessimistic strategies for operations like locking and commits. In optimistic strategies, the data modified by a transaction will be written before the commit occurs, making the commit fast, but harder to undo. The opposite strategy is pessimistic and rarely used, but works well where operations are unreliable and have a lower success rate [52]. The CAP theorem proposed be E. Brewer claims that a distributed system can only guarantee two of the following three properties: Consistency, Availability, and Partition Tolerance [53]. Although, Brewer did later argue that the consistency, availability and partition tolerance exist on a spectrum and not as a binary choice [54]. This indicates that the CAP theorem two out of three rule is oversimplified and choices are context dependent such as the choice between consistency and

availability depends on the system requirements [54]. To clarify the CAP theorem terms, consistency refers to all nodes in the system having the same data and the clients will see the same data at the same time. Availability means that any request for data gets a response without exception, even if one or more nodes are down. Partition tolerance is the ability of the system to continue to operate despite any number of communication breakdowns (partitions) between nodes [53].

Chapter 3

Methodology

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein

3.1 Research method

The research method used in this project is quantitative. Quantitative methods are used to measure the outcome of the new code introduced in the system, how the development process was executed and how the resulting system compares to the original system.

This project revolves around investigating the domain of an ongoing project in a large international collaboration, and to create software to contribute and enhance the processing of jobs. A research method was not immediately chosen as the practical exploration of the domain was the primary focus of the work. Concurrently I started to develop a new algorithm and optimizers to replace the original system. The approach to designing the software was influenced by my existing experience with the DevOps practices and philosophy. DevOps is a modern software development method that focuses on the inter-dependencies between development and operations in the software development process. It builds upon and extends the Agile methodology to support teams in working faster with an iterative approach and deploying

code more frequently [55]. Although DevOps is a suitable set of practices for business, this project requires a more formal methodology to ensure that the research questions are answered satisfactorily. Therefore a research method that could be used to develop software and evaluate the results was required. Design Science was chosen as a research method based on previous experience, and motivated by the requirement to iterate quickly and often to create a solution that could be deployed in a production system. Design Science is well suited for this project as it required me to work closely with domain experts to understand the problem within a complex domain, iterate and evaluate to contribute to the project. The contributions includes a new bookkeeping service in the form of a new algorithm and a set of optimizers to execute the functionality of aforementioned service and other functions. Furthermore, a detailed state diagram was designed to visualize all state changes of jobs in the system. See appendix A and D on pages 106 and 119 for the full source code and diagrams.

Design science is a research methodology that can be employed when researching Information Systems [56]. The methodology focuses on three cycles of activities; *The relevance cycle*, *the rigor cycle*, and *the design cycle*. The main objectives of this methodology is to create a viable artifact and contribute to the knowledge base. Figure 3.1 shows the three cycles in design science research. The cycles bridges the contextual gap between the environment, design science research and the knowledge base. Figure 3.1 further informs that the environment of the research project includes the application domain, organizational and technical systems, people and opportunities & problems within the environment. *The relevance cycle* overlaps the environment and design science research activities and involves the system requirements and field testing. *The design cycle* iterates between the main activities in creating artifacts, and the research process. *The rigor cycle* is the final cycle and is concerned with grounding the research in the existing knowledge base and foundations to add new knowledge to the field [56]. These rigorous methods are involved in the evaluation and construction of the design artifact [57]. Hevner et al. further states that to be effective, design science research must provide clear contributions in the areas of the design artifact, design construction knowledge, and/or knowledge of design evaluation [57]. The guidelines for design science research are presented in Table 3.1.

3.1.1 Design science

Hevner states that "Good design science research often begins by identifying and representing opportunities and problems in an actual application environment" [56]. In this project, the particular research method was retrofitted to the project as I started to develop software before the research method was chosen. The resulting design artifact was of great consequence in the project and would confirm if the proposed software solution was successful. Furthermore, the design artifact would provide the answer to the initial problem statement when deployed and field tested in production.

The full set of requirements of the project had to be discovered through investigation and collaboration with domain experts. The requirements were not fully known at the start of the project, although the problem was established. Learning the domain and the technical systems, moreover to get to know the people that are a part of the environment was essential to create a viable artifact. Having an effective design requires the knowledge of both the solution domain and the application domain [57]. During the first few months of the project, efforts were focused on the relevance cycle, investigating the application domain, and uncovering the true requirements. These requirements had to be uncovered at a significant level of granularity through the *search* process, in which the knowledge could be used in design of the software. The process of design science is inherently iterative and search is a key component of the process to discover an effective solution to a problem [57]. Simon describes the design process as a series of Generate / Test cycle [58]. Where an alternative solution is generated and tested against the requirements and constraints to determine if the solution is viable [58]. Figure 3.2 illustrates this Generate / Test cycle process. The design cycle was initiated when parts of the requirements were known. A consequence of this, was that the first cycles of design naturally went through multiple iterations and evaluations back and forth between the relevance- and design cycle. Primarily between me and technical domain experts at CERN. Before and after design artifacts were created they underwent rigorous field-testing in the form of simulations and manual testing in isolated environments on a local machine. When the design artifacts were ready to be tested in the JAliEn system, they were deployed and tested on production data while writing the results to the development database. This allowed me, in collaboration with technical experts in the organization, to further evaluate and verify the capabilities of

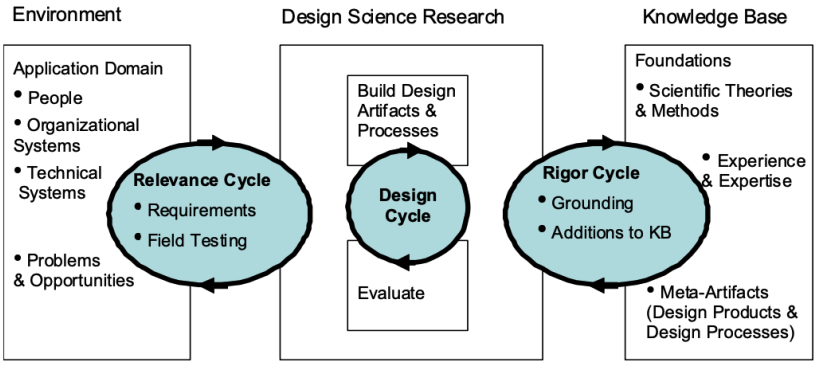

Figure 3.1: Hevner's Design Science Research Cycles [56, p. 2]

the design artifacts. The project has created a valid design artifact that has been deployed in the production and has helped improve the software used in the system. I do not claim to have made any groundbreaking scientific breakthrough in the field, but the work has contributed to the knowledge base of the specific system. Moreover, the JAliEn code has been improved by eliminating frequent long locking time problems and by enhancing the fairness for users. The flow of job states in the system has been simplified and flow diagrams have been created which covers every state change in the system. Furthermore, the algorithms that are presented in sections 3.4.1 and 3.4.2 leverages knowledge that are well established mathematical methods, an example is linear interpolation. These new discoveries and creations are grounded in the foundation of existing scientific theories and methods, a process that is executed through the rigor cycle. How the design science research cycles were applied to the JAliEn system is illustrated in Figure 3.3.

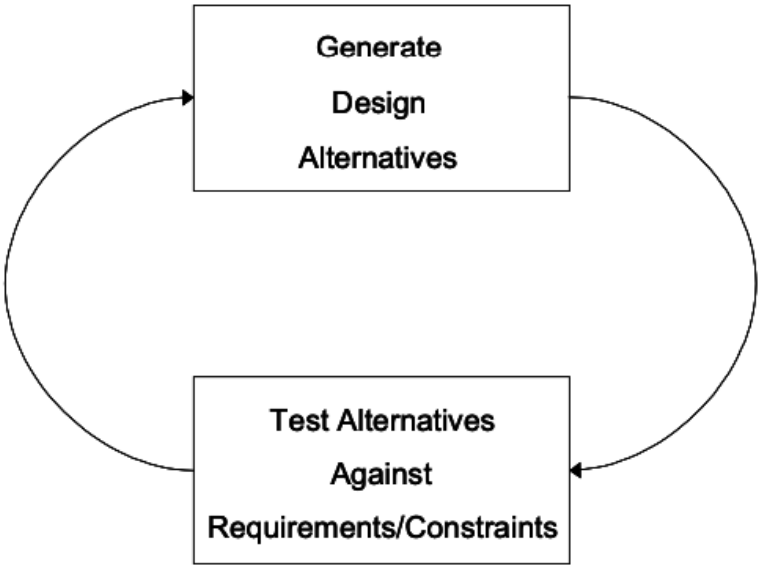

Figure 3.2: Simon's Generate / Test cycle [58] by Hevner et al. [57, p. 15]

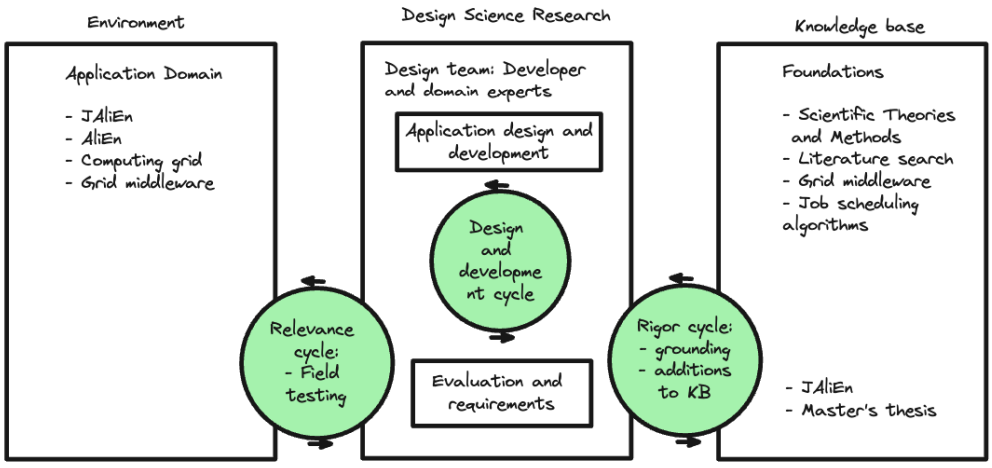

Figure 3.3: The Design Science Research Cycles applied to the JAliEn

Guideline	Description
Guideline 1: *Design as an Artifact*	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: *Problem Relevance*	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: *Design Evaluation*	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: *Research Contributions*	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations and/or design methodologies.
Guideline 5: *Research Rigor*	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: *Design as a Search Process*	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: *Communication of Research*	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Table 3.1: Hevner et al.'s guidelines for Design-Science Research [57, p. 83]

3.2 Application development and the process

As described in section 3.1.1, the start of the project involved investigating the application domain and uncovering the requirements for the project. The full requirements for the final solution were not known at the beginning of the project, neither by the domain experts nor by me. The experts provided general ideas about what the problems were and what the requirements to solve them could be. I spent weeks investigating the old system and its code base to uncover sufficient information to start developing the new service. The investigation is described in section 3.3. I started to develop the new service shortly after the original algorithm was understood. The initial stages

of development focused on investigating and recreating the legacy algorithm, as well as developing a test suite to simulate its behavior. The same test suite would be used to compare the old to the new algorithm that was created to replace the old algorithm. As development progressed over several months the algorithm was tested and the requisite optimizers were implemented to support the algorithm and provide the service. Smaller parts of the system were deployed to production when they were ready to be tested on real data to see what the resulting numbers were and how the database behaved. By December 2023 the first version of a deployable algorithm and optimizers were all deployed to production, however writing to the development database. Most new changes were deployed frequently. Subsequently, in collaboration with the domain experts we identified an additional issue that caused the database to lock up. Moreover we identified more optimizers that needed to be reimplemented to fully replicate the functionality of the optimizers in AliEn, the legacy system. Over the coming months I rewrote the algorithm and several more optimizers to bring all the functionality of the previous system to the new system. In the end of April 2024 the algorithm and its supporting optimizers were the first to be deployed to production and replaced parts of the previous system. Less than two weeks later, the next round of optimizers were deployed and these changes solved the last of the long locking duration problems. Over the coming weeks we discovered a few issues which required ameliorating. In June 2024 the issues were remedied and the remaining optimizers were deployed to production.

I conducted the development of the JAliEn software by writing code in Java with JDK 11 [59] and using the open source build automation tool Gradle [60]. Furthermore, this process also involved a significant number of MySQL queries, initially utilizing MySQL 5.7. This was later upgraded to MySQL 8 by the domain experts in February 2024. JAliEn uses GitLab as a version control system and I used GitLab to manage this and the AliEn project during development.

Additional software used:

- IntelliJ IDEA [61] - Integrated Development Environment (IDE) for Java
- Datagrip [62] - Database management tool used to write and execute SQL queries

- Excalidraw [63] - Diagramming tool used to create flow diagrams
- GitHub Copilot Chat [64] - Experimental AI based software used to understand the original code base
- Python 3 [65] - Used to plot graphs and analyze data
- Kaggle [66] - Used to execute Python code and share results

The entire source code of the project with emphasis on my work, is made available as attachments, described in Appendix A.

3.3 Investigating the original code

The premise for the project was to investigate the existing code of the bookkeeping service in the legacy AliEn Grid middleware. In addition to the optimizers that make up the service, an algorithm is used to calculate the *computedPriority* of users. The *computedPriority* value is the result of the calculation of multiple input parameters and is used to determine which user is prioritized to execute their job next. Access to the legacy code base was granted by another developer and the code was uploaded into a personal GitLab repository. The code was analyzed both manually and with the aid of an experimental version of the artificial intelligence based software GitHub Copilot Chat [64]. This was to identify and to understand the structure and functionality of the service. The old code was written in Perl, a general-purpose programming language [67] which was inspired by C. The database used is MySQL and the queries are written in SQL.

Most queries created in the code are built by utilizing the Java StringBuilder class to create prepared statements. A prepared statement is a query without actual values that the database server can cache the planning of, recycling it between repeated invocations and only changing the numerical values or any other dynamic parameters that the code supplies [68]. Data classes representing different job states are used instead of supplying integer values directly in the query. This approach to building queries makes the code more resilient to possible changes in the meaning of domain specific values. The AliEn code has multiple services and the optimizers are split into three categories; Job, Transfer and Catalogue. The first category is relevant for this project as the Transfer category is already fully implemented in the new

system, JAliEn. The Catalogue category is not relevant and, therefore, is outside the scope of this project. The algorithm in the AliEn code base had to be identified to recreate and improve it for JAliEn. A backup of the legacy algorithm is in the *Taskqueue_preTokens.pm* file at line 2193 [69] as a part of an update query on the PRIORITY table.

3.4 The priority calculation algorithm

The following paragraph outlines the core values upon which the algorithm is built. These values are derived from the original Perl code of AliEn. The priority value calculated by the legacy algorithm had a minimum of 1 if the user had exceeded certain thresholds, but the computed value could reach into the thousands, with higher values indicating higher priority. The legacy algorithm, existing within a SQL query, is displayed in Listing 3.1 starting at line 16, which displays the query and algorithm used by the legacy service. The formula consists of two conditionals, one nested within the other. The first conditional is to verify if the quota for maximum quantity of parallel jobs is exceeded, if within the limits the next conditional is checked. The quotient of running jobs and the maximum amount of parallel jobs running is the current *userload* on the system, this is used in the second conditional. This *userload* value has no strict interval, but depends on the magnitude of the input parameters. The value *(2-userload)* as found in the legacy algorithm, is multiplied with *priority*, the baseline priority of the user, and compared to verify if the value is greater than 0. In the event of both conditionals being true, a new priority is calculated, this value is referred to as the *computedPriority*. Should either conditional yield a false outcome, the value of 1 will be assigned to *computedPriority*, representing the minimum possible value determined by this algorithm. The value of priority is in the magnitude of up to 20000, which is significantly higher than the other values in the formula. This results in the baseline priority having a large impact on the *computedPriority*. The formula does not account for multi-core processing of jobs and treats each job as if it only uses a single core of CPU. Furthermore, the algorithm does not significantly punish the utilization of large amounts of CPU time and cores, which can cause users with low baseline priority to not be prioritized to execute their jobs because their *computedPriority* is too low compared to other users.

Parameter	Min	Max	Explained	Is Better
userload	0	< 1	$= running/maxParallelJobs$	Lower
priority	1	20000	Baseline priority	Higher
running	0	$user.maxParallelJobs$	activeCpuCores	Lower
maxParallelJobs	20	200,000	Quota: max CPU cores in use	Higher
computedPriority	1	$< 10{,}000$	Calculated value	Higher

Table 3.2: Parameters and intervals for the legacy algorithm. The *Is Better* column indicates if a lower or higher value is better for the user when calculating a *computedPriority*. See Listing 3.1 for the usage in the legacy algorithm.

An important area to improve the original algorithm is to ameliorate the fairness of the algorithm. Specifically meaning that enhanced fairness ensures equitable allocation of CPU time on the Grid to execute jobs for all users, regardless of their baseline priority. Users who have not executed jobs in the last 24 hours, should have the opportunity to run their jobs. Thereafter, the resulting *computedPriority* ought to decrease similarly for all users regardless of baseline priority, except the production users such as Aliprod which are highly important for the processing of physics data from the experiment. These production users should in practice always be able to run their jobs, but by ameliorating fairness the less active users will be allocated resources to execute jobs.

```
$self->do(qq{
    UPDATE PRIORITY p
    LEFT JOIN (
        SELECT userid, COUNT(*) w
        FROM QUEUE q
        WHERE statusId=5
        GROUP BY userid
    ) b USING (userid)
    LEFT JOIN (
        SELECT userid, COUNT(*) r
        FROM QUEUE q2
        WHERE statusId IN (10,7,11)
        GROUP BY userid
    ) b2 USING (userid)
    SET waiting=COALESCE(w,0), running=COALESCE(r,0)
       ,
        userload=(running/maxparallelJobs),
        computedpriority=(
            IF(running<maxparallelJobs,
                IF((2-userload)*priority>0, 50.0*(2-
                   userload)*priority, 1),
                1)
        )
});
```

Listing 3.1: Original priority update query and algorithm

3.4.1 Modifying the legacy algorithm

The original algorithm was rewritten in Java to facilitate the implementation of a new algorithm. The Java version was written to mimic the original algorithm by maintaining the semantics of the original code. Furthermore, this implementation was used as a comparison to establish a baseline for the resulting *computedPriority* values when executing simulations in a controlled environment to compare the algorithms.

The new version of the algorithm built upon the original by extending the functionality and adding additional parameters to the formula. Moreover,

a conditional check to assure that the number of jobs and resource quotas of the user has not been exceeded was added to the start of the method. Furthermore, several of the variable names, and the meaning of them, have been changed to accommodate the updated functionality. For example the *running* and *maxParallelJobs* variables have been renamed to *activeCpuCores* and *maxCpuCores* respectively. This describes that a single job no longer only count as one CPU core spent, rather the number of CPU cores requested by the job. Similarly, the *maxParallelJobs* has been renamed to reflect this modified definition, and the quotient of these values are equivalent to the *userload* parameter. Although the column names remain the same in the PRIORITY table, the underlying meaning in the code has been updated to reflect the new functionality.

```
    private static void updateComputedPriority(
       PriorityDto dto) {
        if (isQuotaExceeded(dto)) {
            return;
        } else {
            int activeCpuCores = dto.getRunning();
            int maxCpuCores = dto.getMaxParallelJobs
               ();
            long historicalUsage = dto.
               getTotalRunningTimeLast24h() / dto.
               getMaxTotalRunningTime();

            if (activeCpuCores < maxCpuCores) {
                double coreUsageCost =
                   activeCpuCores == 0 ? 1 : (
                   activeCpuCores * Math.exp(-
                   historicalUsage));
                float userLoad = (float)
                   activeCpuCores / maxCpuCores;
                dto.setUserload(userLoad);
                double adjustedPriorityFactor = (2.0
                    - userLoad) * (dto.getPriority()
                    / coreUsageCost);

                if (adjustedPriorityFactor > 0) {
                    dto.setComputedPriority((float)
                       (50.0 *
                       adjustedPriorityFactor));
                } else {
                    dto.setComputedPriority(1);
                }
            } else {
                dto.setComputedPriority(1);
            }
        }
    }
```

Listing 3.2: Modified legacy algorithm to calculate *computedPriority*

The second conditional from the original algorithm has been extracted into the variable *adjustedPriorityFactor* and received a modification to include a cost for the amount of CPU cores used recently. This variable is derived from the historical resource usage over the preceding 24 hours. Calculated as the quotient of total running time and the maximum total running time for that user in the last 24 hours, the *historicalUsage* variable is created.

Parameter	**Min**	**Max**	**Explained**	**Is Better**
userload	0	< 1	$= running/maxParallelJobs$	Lower
priority	1	20000	Baseline priority	Higher
running	0	$user.maxParallelJobs$	activeCpuCores	Lower
maxParallelJobs	20	200,000	Quota: max CPU cores in use	Higher
computedPriority	1	$< 10{,}000$	Calculated value	Higher
totalRunningTimeLast24h	0	$user.maxTotalRunningTime$	CPU time spent	Lower
maxTotalRunningTime	1×10^7	1×10^{14}	Quota: max CPU time accumulated	Higher

Table 3.3: Parameters and intervals for the modified legacy algorithm. The *Is Better* column indicates if a lower or higher value is better for the user when calculating a *computedPriority*. See Listing 3.2 for the usage in the modified algorithm discussed in this section.

Furthermore, this variable is used in the calculation of *coreUsageCost* which will be 1 if the user has been inactive recently, or it's the product of *activeCores* and *Math.exp(-historicalUsage)* for any active user. *Math.exp* was chosen to utilize a function that would work for both small and very large numbers, *Math.log* was attempted, but *Math.exp* worked better for the intended purpose. The *Math.exp* function returns e raised to the power of a double value [70], and is used with a negative of *historicalUsage* as the parameter to avoid massive numbers in the product of the *coreUsageCost* variable. This was required as the *historicalUsage* values can reach a magnitude of billions. Snippets of the algorithm to calculate *coreUsageCost* and *adjustedPriorityFactor* are shown below, and the full algorithm can is displayed in Listing 3.2. The *dto* referred in this section refer to the data transfer object, based on the pattern with the corresponding name [71], that was used to pass the user data to the method.

$$\text{coreUsageCost} = activeCores \cdot e^{-historicalUsage} \tag{3.1}$$

$$\text{adjustedPriorityFactor} = (2.0 - userLoad) \times \frac{dto.getPriority()}{coreUsageCost} \tag{3.2}$$

The *computedPriority* is then calculated as the product of *50.0* (the same value used by the legacy algorithm) and the *adjustedPriorityFactor*.

The flow of this approach is similar to the original, however the new algorithm has been extended with new functionality and parameters. Additionally the code is optimized by the use of variables instead of repeating the same calculations multiple times. Firstly, the algorithm checks if the user has exceeded their quota on computational resources in the last 24 hours, and if the quota for active CPU cores has been exceeded. If the quota is below the threshold, the method will return false and the *computedPriority* will be calculated for the user, otherwise the *computedPriority* value will be set to -1.

Contingent upon the quota being within the limits the parameters for the algorithm are assigned to the aforementioned variables. Predicated that the active CPU cores are below the threshold, the *coreUsageCost* is calculated and the *adjustedPriorityFactor* is determined. If the *adjustedPriorityFactor* is equal to or less than zero, then the *computedPriority* is set to the minimum value of *1*. This could happen if a user is above their quota on the number of cores in use. Otherwise, the *computedPriority* is calculated and the DTO is updated. The method as seen in Listing 3.2, is designed to update the parameter it receives and acts upon a single user at a time.

3.4.2 Design and implementation of a novel user scheduling algorithm

Developing an algorithm from scratch represents a distinct approach compared to refining an existing algorithm with known flaws. By designing the algorithm anew, I had greater flexibility to innovate and construct a formula that more effectively addresses the existing issues, incorporates the desired functionality, and enhances fairness for all users. The following paragraphs will refer to multiple columns and values from the PRIORITY table in the central database, an overview of the table is displayed in the Appendix, see Table C.3 (p. 116).

Similarly to the quota check in section 3.4.1, the *computedPriority* of a user is set to -1 if the quota is exceeded. In addition to comparing the quota for CPU cores in use and CPU time spent in the last 24 hours, this approach also includes the *cost* in the last 24 hours. The *cost* is calculated as the product

of elapsed real time, the number of CPU cores utilized and the price of the job. The *cost* value, or *totalCpuCostLast24h*, is specific to the user and will be increased as the batch jobs of the user are completed. If either of these quotas are exceeded, the *computedPriority* is set to -1 and the method exits. Otherwise, if all conditions are *false* the user is within quota limits and the *computedPriority* is calculated.

$$\text{cost} = \textit{(core * walltime) * price} \tag{3.3}$$

Parameter	**Min**	**Max**	**Explained**	**Is Better**
priority	1	20000	Baseline priority	Higher
activeCpuCores	0	$user.maxParallelJobs$	CPU cores in use	Lower
maxParallelJobs	20	200,000	Quota: max CPU cores in use	Higher
totalCpuCostLast24h	0	$user.maxTotalCpuCost$	$((core * walltime) * price)$ last 24h	Lower
maxTotalCpuCost	10	1×10^{14}	Quota: max total cost	Higher
computedPriority	-1	10 (1 without boost)	Normal interval [0-1]. $-1 = quota$	Higher
boost	0	10	Add boost if $totalCpuCostLast24h < 1M$	Higher

Table 3.4: The input parameters for the novel user scheduling algorithm and their intervals. The *Is Better* column indicates if a lower or higher value is better for the user when calculating a *computedPriority*. Refer to Table 3.5 for the weights used in the calculation of the *computedPriority* and see Listing 3.4 for the usage of the parameters and weights in the algorithm on pages 60 and 62.

The novel user scheduling algorithm is designed to prioritize users that have not used any resources in the last 24 hours. Minimum and maximum thresholds were established for *cost* usage in the previous 24 hours to determine eligibility for a boost. These thresholds are not derived from the user's individual *maxTotalCpuCost* property, but are specifically set for this method to calculate a boost value, ensuring fairness in the process. These minimum and maximum values are set to 100,000 and 1,000,000. If a user is above maximum *cost* usage limit they will not receive a boost. If below minimum, they receive 10 in boost. Linear interpolation is leveraged to find the slope,

$$\text{slope} = \textit{(noBoost - maxBoost) / (maxCost - minCost).} \tag{3.4}$$

which is used to calculate the boost value for users between minimum and maximum. The resulting boost is a rounded to a integer value of:

$$\text{boost} = \textit{maxBoost} + \textit{slope} * (\textit{dto.getTotalCpuCostLast24h()} - \textit{minCost}). \tag{3.5}$$

The resulting boost value is determined by the current usage of resources, this ensures that users with no or little resource usage in the previous 24 hours are prioritized. The calculation to find the boost value is found in the code in Listing 3.3 (p. 61). The *computedPriority* calculated by this algorithm ranges between 0 and 1 when no boost is applied, with higher values indicating higher priority. See the code for the novel algorithm in Listing 3.4 (p. 62). Any boost a user receives ranges from 1 to 10 and is added to the *computedPriority* value after calculation. If a user is ineligible for a boost, the method returns a value of 0. Adding a boost value for eligible users guarantees that users with low cost usage in the last day will have higher *computedPriority* than other users until they spend more resources. The following two equations are an example of how the boost is calculated for a user. The boost value is added to the *computedPriority* value in line 16 in the code snippet displayed in Listing 3.4 (p. 62). The values for the slope are equal in all cases, but the boost value will differ based on the *totalCpuCostLast24h* value.

$$\text{slope} = \textit{(0 - 10) / (}1{,}000{,}000 - 100{,}000\textit{)} \tag{3.6}$$

Example value for *totalCpuCostLast24h* = 350,000:

$$\text{boost} = \textit{10} + -0.0000111111111111 * \textit{(}350{,}000 - 100{,}000\textit{)}. \tag{3.7}$$

In this example the resulting boost value 7,222 is rounded to the closest *long* by using Math.round(), thereafter the boost value is added to the *computed-Priority* value for the user. The code to calculate the boost value is displayed in Listing 3.3.

To calculate the *computedPriority* value the chosen approach is to use a weighted sum of multiple input parameters to determine resulting value. Three weights are used to calculate the *computedPriority* value, these are *cost*, *activeCpuCores* and *priority*. These weights represent key parameters, including the user's baseline priority, similar to the original approach, while

incorporating resource usage through *cost* and *activeCpuCores*. The sum of the weights is 1, allowing for flexibility in adjusting the influence of each value and providing the option to add additional parameters in the future. The weights and their magnitudes were chosen in collaboration with the domain experts. These weights, along with normalized quotients, are used by the algorithm to compute a weighted sum, which is then added to the boosted value to calculate the new *computedPriority*. The specific weight assignments are as follows, with higher values indicating a greater weight for each parameter:

Parameter	**Weight**
cost	0.1
activeCpuCores	0.5
priority	0.4

Table 3.5: Relative weights of different parameters considered by the novel algorithm

The cost quotient is normalized by utilizing the square root of total cost (*totalCpuCostLast24h*) used, divided by the maximum cost (*maxTotalCpuCost*) for the user. The activeCpuCores quotient is normalized by dividing the active CPU cores by the maximum CPU cores for the user. The normalized priority quotient is the difference of 1 - the baseline priority for the user divided the maximum priority value of any existing user. Each distinct weight is multiplied with the correlated normalized quotient, and the sum of these values is added to the boost value to calculate the *computedPriority*. The value which is returned to the caller is the absolute of *computedPriority* -1. This ensures that the value is positive and that the value decreases in relation to resources spent. The code for the user scheduling algorithm is displayed in Listing 3.4.

```
    protected static int findBoostValue(PriorityDto
       dto) {
        int noBoost = 0;
        int maxBoost = 10;
        int minCost = 100_000;
        int maxCost = 1_000_000;

        if (dto.getTotalCpuCostLast24h() > maxCost)
           {
            return noBoost;
        }

        if (dto.getTotalCpuCostLast24h() < minCost)
           {
            return maxBoost;
        }

        // Linear interpolation to calculate the
           boost value
        double slope = (double) (noBoost - maxBoost)
            / (maxCost - minCost);
        return (int) Math.round(maxBoost + slope * (
           dto.getTotalCpuCostLast24h() - minCost));
    }
```

Listing 3.3: Method to find the boost value to *computedPriority* for enhanced fairness for users. This method is invoked by the novel user scheduling algorithm in Listing 3.4. Further explanation in Section 3.4.2.

```
    private static void updateComputedPriority(
       PriorityDto dto) {
        if (isQuotaExceeded(dto)) {
            return;
        }

        float maxBaselinePriority = dto.
           getHighestPriority();
        float costWeight = 0.1f;
        float activeCpuCoresWeight = 0.5f;
        float priorityWeight = 0.4f;

        float costQuotient = (float) Math.sqrt(dto.
           getTotalCpuCostLast24h() / dto.
           getMaxTotalCpuCost());
        float normalizedRunningQuotient = (float)
           dto.getRunning() / dto.getMaxParallelJobs
           ();
        float normalizedPriority = 1 - (dto.
           getPriority() / maxBaselinePriority);

        float weightedSum = costWeight *
           costQuotient + activeCpuCoresWeight *
           normalizedRunningQuotient +
           priorityWeight * normalizedPriority;

        float computedPriority = (weightedSum +
           findBoostValue(dto));

        dto.setComputedPriority(Math.abs(
           computedPriority - 1));
    }
```

Listing 3.4: Novel user scheduling algorithm which is deployed on the central instances of the JAliEn in production. This algorithm is further explained in Section 3.4.2.

3.5 Optimizers

Optimizers are asynchronous services that are executed on the central machines and interact with the database and primarily is tasked with synchronizing information between database tables and the central services and worker nodes. An optimizer will run using a dedicated thread on a periodic interval, similar to a cron job, and execute a set of tasks. Furthermore, optimizers are a significant part of the functionality in JAliEn and are involved in a multitude of different tasks, examples are to transfer a batch job state, inserting jobs if the Job Description Language (JDL) file requirements are met, deliver monitoring information, and splitting larger jobs into smaller ones.

In the AliEn code the optimizers are split into three categories; Job, Transfer and Catalogue. The Job optimizers are responsible for handling job states and job information, the Transfer optimizers are responsible for transferring files between sites, and the Catalogue optimizers are responsible for handling metadata and file information. The optimizers in JAliEn are responsible for similar tasks, but my work focused on the Job optimizers. The Job optimizers are responsible for updating the job states, calculating the *computedPriority* of users and purging old jobs from the database. The original optimizers were not optimal as they caused frequent issues with processes waiting for a long time to access the database. A major reason for this is the size and complexity of the database queries executed by those optimizers. Specifically, there were two optimizers which caused the most frequent problems, these were the ones calculating and updating the *computedPriority* of users in the database tables and the bookkeeping optimizer which counted the number of jobs in a particular state and summarizing the total cost for each site. I have reimplemented these optimizers to avoid the long locking time and to improve the performance of the system. Furthermore, these new optimizers benefit from using the isolation level *READ UNCOMMITED* to perform SELECT statements in a nonlocking fashion. The risk here is that an earlier version of the row might be used, otherwise this isolation level works similar to *READ COMMITTED* [72]. In this context a slight inconsistency is acceptable as the information is frequently reconciled by two of the optimizers.

To mimic and improve upon functionality of the original bookkeeping service, in addition to the new user scheduling algorithm, a suite of new optimizers has been developed. All the optimizers run on the central instances of JAliEn

and they extend the *Java.lang.Thread* class. Each individual optimizer starts its own thread that is executed on periodic intervals. These optimizers are executed on one or several central instances of JAliEn and work together as a set of methods that aims to ensure that information is synchronized across the system. Information about user priority, CPU time usage and cost in addition to job status are being calculated, tracked and updated by the optimizers by persisting values in the database. In addition to this, the optimizers are responsible for purging old jobs to ensure that the database is not filled with outdated job entries.

JAliEn has a custom database synchronization utility class that can be used to ensure that a certain optimizer only executes on a single instance of the central services, even if there are multiple instances running. This prevents any instance of the same class to execute the task if it already has been started, or completed it within a frequency interval by another instance. The frequency interval is specific to each optimizer and the duration equals the sleep period for each optimizer thread.

The optimizers that I created are described in detail in the subsequent sections.

3.5.1 How the novel algorithm and optimizers are used in the system

The novel algorithm is only triggered when called with a parameter specifying whether all users or only active users should be recalculated. All the optimizers extend the Java *Thread* class and can be started in the same manner as a thread. This dedicates a thread to each optimizer, which is executed on its own interval. The parent class of all optimizers holds an array of available optimizers and starts them, by category, through a *run()* method. Whenever an optimizer finishes the task, it will sleep for a predetermined interval before the next execution. Most optimizers are executed on a single instance of the JAliEn central services, as all instances update the same database. Whether a central instance should run an optimizer or leave it for another, is predicated upon a boolean check to the database to verify the frequency since last execution. Furthermore, if one interval between executions have elapsed and the central service is the first one to ask, it will be the one to execute the optimizer. This is to ensure that the optimizer is only executed once, and

that the database is not locked by multiple instances trying to execute the same optimizer. The exception to the rule is the *PriorityRapidUpdater* that is executed on all central instances as this needs to maintain a registry in memory and keep a count of job state changes.

3.5.2 JobAgentUpdater

The JobAgentUpdater runs on a short interval on a single instance of the central services of JAliEn and is responsible to ensure that the *priority* column in the JOBAGENT table is updated. The priority value is based on the user's calculated priority which is stored in the *computedPriority* column in the PRIORITY table. The update is executed by inner joining PRIORITY with JOBAGENT on *userId* and setting the priority value in JOBAGENT to the value of *computedPriority*.

```
UPDATE JOBAGENT INNER JOIN PRIORITY USING(userId
    ) SET JOBAGENT.priority = PRIORITY.
    computedPriority;
```

Listing 3.5: JobAgentUpdater query

3.5.3 PriorityRapidUpdater

On the computing Grid there are numerous computing nodes on sites around the world that are available to run jobs. To guarantee a fair environment where every user is provided CPU time, it is crucial to track resource usage data. Whenever a batch job completes, the site where the job was executed needs to record resource usage about CPU time and *cost*. The *cost* is calculated as the product of the number of CPU cores utilized, wall-clock time, and the price of the job. The *price* is a value set by the user in the Job Description Language file when submitting a batch job, to indicate perceived importance of the job. Additionally, this *price* value is used to sort between jobs by a given user and the total price for a job is calculated by a different optimizer, when the job reaches the *DONE* state. The resource information is stored in a ConcurrentHashMap in memory which acts as a global registry, containing each user and the resource usage for recently active users on that node. Information about how much computing resources a user has used is important when calculating the *computedPriority*. These values are

required in the updated algorithm to evaluate user quotas and to provide fairness by determining whether a user should receive an increment to their *computedPriority*. The job usage information, and the running and waiting jobs on each site, are stored in the aforementioned registry. Users and their usage data is stored in the registry when a state change is initiated in the application. These state changes can be a job being started and moved from *waiting* to *running*, or a job could be stopped due to an error or kill order. A *running* batch job that completes execution will attempt to transition to a final state if it passes validation.

To maintain accuracy, the optimizer flushes the registry to the database. In the registry, correctness is essential to accurately report resource usage in a timely manner. To ensure all changes in the registry are captured, a deep copy of the registry is made, allowing any updates made during the flush process to be included. All the variables in the registry are atomic, and are updated using atomic operations. The registry is a ConcurrentHashMap which supports concurrency for read operations. Furthermore it blocks other writes to the same key, while allowing writes to other keys, this provides high concurrency for update operations [73].

The PriorityRapidUpdater optimizer is executed on all the central instances of JAliEn. This optimizer has three main responsibilities. The first task is to identify users which have all values in the registry set to zero, and remove these inactive users from the registry. The second task is to update the PRIORITY table columns *waiting*, *running*, *totalRunningTimeLast24h* and *totalCpuCostLast24h* for active users with the values stored in the registry. *Active users* are defined as those who have utilized computational resources on the computing Grid within the past 24 hours. *Inactive users* refer to all other users who have not used resources during this time frame. Moreover, the *active* flag in the same table is set to *true* when the optimizer updates user values. When persisting data to the database, the stored value will be the greater of either the snapshot value or 0. The final task is to subtract the updated values from the registry as they have been persisted. To ensure that no changes occured to the registry while the optimizer was updating the PRIORITY table, a snapshot of the registry was taken before the update was initiated. Values from the snapshot are then subtracted from the values in the registry. To obtain an accurate snapshot, a deep copy is taken to avoid the copy and the original to share location in memory. The deep copy uses a constructor in the nested static class *JobCounter*, that holds the values of

the registry. The *JobCounter* constructor safeguards that new instances of atomic variables are created based on the original value in the registry.

3.5.4 PriorityReconciliationService

One of the most critical optimizers is the PriorityReconciliationService, which is executed periodically and in exclusive mode on one of the central service instances. The primary task of the PriorityReconciliationService is to ensure that user activity data is synchronized across tables. When batch jobs running on worker nodes on sites are completed, they report resource usage information to the central database. This information is utilized when calculating *computedPriority* for users. To optimize the update operations, the update operations are split into two separate queries. The first operation updates the active users, while the second adjusts the values for all users. The first operation is executed more frequently and completes in a shorter time, as explained in Section 3.5.5. The second operation, managed by the PriorityReconciliationService optimizer, runs less frequently and requires more time. This ensures that all users' values are updated, regardless of their activity, and that their data remains accurate in the event of batch job submissions.

The first operation of the optimizer is to execute a query that selects *cpucores*, *statusId*, *cputime* and *cost* from users that have shown activity in the last day in the QUEUE and QUEUEPROC tables. Activity means that the user has used computational resources within the last 24 hours. The users are mapped by id, where *cputime* and *cost* is summarized for each user. The number of active CPU cores in use is accumulated, and the *statusId* is filtered to find the total of jobs in the WAITING state. Thereafter the values are persisted in the PRIORITY table. Subsequently a new select query is executed on the PRIORITY table to retrieve all users with values larger than zero in the *totalCpuCostLast24h* and *totalRunningTimeLast24h* columns. The resulting set from the query is compared against the content from the first query for active users to verify which users are still active. For all of the non-active users the two aforementioned columns are reset to 0. The final task of the optimizer is to trigger the recalculation of *computedPriority* for all users in the PRIORITY table. This is triggered through the *CalculateComputedPriority.updateComputedPriority()* method, see Listing 3.2. Subsequently the optimizer will sleep for an hour before the next execution. This optimizer

is executed infrequently and updates values for all users in the PRIORITY table.

3.5.5 ActiveUserReconciler

A new *active* flag was introduced as a column in the PRIORITY table to indicate whether a user is active or not. Activity would indicate that the user has been executing jobs within the last twenty-four hours. The ActiveUserReconciler optimizer is responsible for updating the *active* flag to false when a user has not spent CPU execution time in the last 24 hours, and the active flag is set to *true*. Furthermore, the optimizer will trigger a recalculation of the *computedPriority* for all active users, which is updated in the PRIORITY table. This optimizer is executed on a single instance of the central services and runs on a regular 5 minute interval. The functionality of the optimizer complements the PriorityRapidUpdater and PriorityReconciliationService optimizers. Firstly by negating the active flag for users that have not used any CPU in the last 24 hours, secondly by prompting the recalculation of *computedPriority* for active users. By updating only the active user's *computedPriority* on a more frequent interval than the PriorityReconciliationService, there are fewer database rows to update which reduces execution and locking time spent. Moreover, it ensures that information in the tables are updated more frequently and the window of opportunity for a user to start more jobs than their quota allows is reduced.

3.5.6 InactiveJobHandler

The InactiveJobHandler optimizer is executed on a single instance of the central services, executing at brief intervals to guarantee that inactive jobs are promptly transitioned to either a ZOMBIE or EXPIRED state. Modifying the status of jobs that have not had a heartbeat within a predetermined timeframe is important to ensure that a job is not kept in a running state when it is not active. Heartbeats are regularly dispatched from the remote job agent node, where the job is deployed, to the central services to indicate ongoing activity. The optimizer will move jobs to a ZOMBIE state if the job has not had a heartbeat in the last hour. If the node has not forwarded the job heartbeat in the last two hours and the job is already in a ZOMBIE state, then it is moved to the an EXPIRED state, which is a final state. The timestamp of the heartbeats are stored within the *lastUpdated* column of the

QUEUEPROC table. Subsequently, the optimizer invokes a specific method from the *TaskQueueUtils* class to transition the job to an updated state, provided input parameter of the current state is unaltered by other processes. The aforementioned class is a preexisting utility class which typically involves database operations. Novel optimizers developed in this project utilize this class to promote code reuse and to ensure synchronization of information when batch job state changes. As of the time of writing, the system does not support tracking of historical job states and therefore cannot move a job in the ZOMBIE state back to its previous running state.

3.5.7 SitequeueReconciler

The SitequeueReconciler optimizer is used to count the number of jobs in a particular state and summarize the total cost accumulated by all jobs for each site. The total cost for a site is the summarization of the accumulated *cost* value from the QUEUEPROC table for all jobs in the last day (see the Appendix Tables C.1, C.4 and C.5 (p. 115, 117 and 118) for the full SITEQUEUES, QUEUE and QUEUEPROC table columns). The optimizer is executed on a single instance of the central services and runs on a regular one hour interval. Initially a select query is joining the QUEUE and QUEUEPROC table to count the number of jobs in every state for a specific site. Furthermore, the same query sums the cost for the respective state on every site and groups the result by site and state. The results from the select query are stored in a list of data transfer objects. The Java stream API is used to group the objects by cost and count into two collections. The cost for each site is stored in a map of site and total cost. The count is mapped for the respective site and status in a map of sites with the value being another map of the state and count. Values from these collections are combined in a single query to update the SITEQUEUES table with a varying number of states and cost per site.

The three subsequent optimizers are responsible to transition job status, purging old jobs from the database, and maintain synchronization among job states.

3.5.8 OverwaitingJobHandler

Ensuring that jobs which have not concluded within an extended period of time are addressed, the OverwaitingJobHandler optimizer has an essential role. Executed every 6 hours on a single instance of the central services the OverwaitingJobHandler optimizer transitions jobs that have been inactive for more than seven days. Furthermore, the optimizer supports a user defined parameter supplied through the job description file. If this parameter is set, the QUEUE table *expires* column will be set and this value will be used instead of the default seven day value. The optimizer transitions inactive jobs from the WAITING state to an ERROR_EW state which is a final state. This optimizer utilizes the same method as the InactiveJobHandler optimizer to transition the job to an updated state.

```
1   private static String getDeleteQuery(Set<Long>
       finalJobs) {
      StringBuilder queueIds = new StringBuilder()
         ;
3     for (Long id : finalJobs) {
        queueIds.append(id)
5               .append(",");
        TaskQueueUtils.putJobLog(id.longValue(),
           "OldJobRemover", "Job to be removed
          by OldJobRemover optimizer", null);
7     }
      if (queueIds.length() > 0)
9       queueIds.deleteCharAt(queueIds.length()
          - 1);

11    String query = "DELETE FROM QUEUE WHERE
         queueId IN (" + queueIds + ") OR split IN
          (" + queueIds + ")";
      logger.log(Level.INFO, "OldJobRemover query:
          " + query);
13    return query;
  }
```

Listing 3.6: Delete job query

3.5.9 OldJobRemover

This optimizer is designed to remove any master, sub, or single job that matches the specified criteria, with each job uniquely identified by a *queueId* stored in the QUEUE table. Master jobs will have one or multiple subjobs associated with them. All subjobs have a *split* value that equals the *queueId* of the master job. The OldJobRemover optimizer runs on a single instance of the central services, and operates on a predetermined interval of two hours to purge jobs that have remained in a final state for a period exceeding five days. This measure is implemented to prevent the QUEUE table from expanding indefinitely. The database query executed by this optimizer retrieves all master and single jobs remaining in any final state for over five days. Subsequently, the deletion query exploits the association between master jobs and their subjobs via the *split* column, facilitating the removal of all subjobs linked to a master job. Furthermore, the optimizer efficiency is enhanced by not selecting subjobs individually, thereby reducing the number of unique identifiers that need to be iterated over. This approach is displayed in the query presented in the Listing 3.6.

3.5.10 MasterSubJobReconciler

In JAliEn there is a possibility that jobs can be resubmitted for execution if the job fails or is killed. This can lead to situations where for example a subjob is resubmitted and is then in a running state, this means that the master job state needs to be updated to reflect this change. The MasterSubJobReconciler optimizer is responsible to ensure that the job status is correct for the master job and its associated subjobs.

This optimizer executes on a single central service instance at regular, brief intervals and is tasked with two primary functions. Initially, upon execution it issues two select queries to the database: One to find master jobs in running states and another to identify those in final states. The optimizer's first function involves verifying whether all subjobs linked to a master job in a running state have reached a final state. Should this condition be met, the master job is then transitioned to a final state. The second function entails confirming that all subjobs associated with a master job in a final state, are also in a final state; if confirmed, no further action is required. The overarching objective of this optimizer is to maintain consistency across the states of a master job and its subjobs, ensuring they do not exhibit

heterogeneity in terms of their operational status. Should any inconsistencies be identified, these inconsistencies are aggregated to a bulk update query. This query is executed to rectify the state of the affected jobs.

3.6 Testing the algorithm and optimizers

To verify the functionality of the algorithm and optimizers before they could be tested in JAliEn in a live environment they were tested in isolation. All approaches to the algorithm, the original and algorithms designed by me, were tested using comparable simulations. These algorithms are described in Sections 3.4, 3.4.1 and 3.4.2. The optimizers described in Section 3.5 were tested on data from the development database to ensure that they performed as expected. The testing of the algorithm and optimizers is described in the following sections.

3.6.1 Testing the algorithm

To test the old algorithm against the new algorithm to evaluate the output values of *computedPriority* and fairness over time, an isolated simulation was created in Java. The simulation was written as a standalone execution outside of the JAliEn code. An environment was created where up to six users would compete to start jobs, until a total of 100,000 jobs were started. Every job could be a multi-core job and the number of cores per job was assigned by a method that utilizes a seeded random number generator. This ensures replicability where either a single core or an even number of cores from two to fourteen is assigned to the job. Experiments were conducted with sets of different combinations of baseline priority and maximum parallel jobs. The input parameters to the simulations for each user were as follows:

- User id
- Active cores in use
- Maximum cores that can be used
- Baseline priority
- Total cost in the last 24 hours
- Maximum cost in the last 24 hours

- Total running time in the last 24 hours
- Maximum running time in the last 24 hours

The output from the simulation was saved to a CSV file for further manual analysis and to create plots. These plots were used to compare changes over time and changes to behaviour when the input parameters vary. The output contained the user id, current number of running jobs, the computed priority and the total amount of jobs started. The simulation was executed with the new algorithm and the original algorithm to compare the results. For each execution of the old algorithm the input parameter for the number of cores was set to 1, as the legacy algorithm does not account for multi-core jobs. The entire simulation was executed in an isolated environment which means that jobs are not actually started, but the user with the highest *computedPriority* is returned for each iteration. The amount of times a user has the highest *computedPriority* is compared between the two versions of the algorithm to evaluate fairness. The results are evaluated in section 4.1.

3.6.2 Testing the optimizers

During development, each of the optimizers was tested locally by starting a new application which invokes the optimizer. Where required, test data was provided and setup for the environment. The JAliEn environment is split into a production database and a development database which is an older replica of production data. The optimizers act on values in the database, therefore to have realistic data to test with, every read operation was executed on the production database while all write operations are targeted to the development database. This provides a safe environment to test the optimizers without affecting the production environment. Furthermore, this makes it trivial to compare values in the development database with the production database to ensure that the optimizers are functioning as intended. Due to the changes introduced by the optimizers, and the new algorithm, the values will differ between the databases. One example is that multiple cores are counted for a single job in the new algorithm. This approach to testing in production allows for a longer test period with real data and computational load on application. Moreover, testing on production data provides a realistic environment to verify if the database lock duration improvements work in an environment with a high number of I/O operations. Furthermore, it allows for a direct comparison of states before and after changes, to verify if

the new optimizers ameliorated the locking issues.

A major indicator to measure is to investigate the number of restarts on the database after the original optimizer is replaced with the new optimizer, the *SitequeueReconciler*. After this optimizer was introduced the database should not be restarting anymore due to locking up and being killed, more on this in Section 4.3. I must clarify that MySQL does have capabilities to detect and handle deadlocks, this is not a classic deadlock case, but it is hindering the database from responding as expected. The domain experts found it more efficient to restart the database when the number of waiting processes exceeded a threshold of 4,000 than waiting for it to recover. Reaching or exceeding this limit was a sign that an optimizer had locked the table for a long time and many processes had started to queue up when attempting to access the database. From experience, they found that it would take too long for the system to recover and the number of processes waiting would continue to rise. The result of this would be that the database would not catch up with the backlog of queries for many hours.

3.7 Evaluating the algorithm and optimizers

The algorithm and several of the optimizers are designed to work together, and to evaluate them they need to be tested together. Supplementary optimizers can be tested and evaluated in isolation to determine if their behaviour is as expected. Evaluation periods of several weeks, followed by adjustments applied to the optimizers were necessary to achieve the desired output. One of the major problems in the old system is the delay caused by many requests queuing up to access a database table that is locked for a long time. The temporary workaround to quickly recover the functionality of the database was to utilize a script to count how many processes were attached to the database. If the number of attached processes or threads exceeded 4,000 the database application would be killed.

The script would log a multitude of information, including the time and cause of the restart which identified the exact query that locked a table and caused processes to queue up awaiting access to the database. Including the time and cause of the MySQL restart was invaluable to compare how the database acted before and after the new optimizers were deployed. This information was used to determine whether an optimizer could reduce the long locking

times and help decrease the total number of restarts, with the ultimate goal of achieving zero restarts. The results from the evaluation of the algorithm and optimizers are presented in Section 4.

The *computedPriority* values were evaluated by comparing outputs from the PRIORITY table in production and development. The values were not expected to be the same, but given a set of users the order of the values should be similar as baseline priority input parameter and quotas are the same. Adjustments were made for the new *computedPriority* value, which was expected to be lower than the original due to the new algorithm penalizing CPU core usage over the last 24 hours. This adjustment helped ensure a more accurate prioritization of users based on recent resource consumption. The fairness of the algorithm was evaluated through simulations and manual inspection. The analysis focused on how the *computedPriority* values diminished in relation to changes in CPU cores in use, as well as the baseline priority value set for each user. This provided insights into the algorithm's behavior under varying loads and helped ensure that priority adjustments reflected both recent resource consumption and user baselines. Following the deployment of the new algorithm into production the values were monitored closely, and configuration for optimizer frequency interval was adjusted to ensure optimal performance and to verify the fairness for all active users.

Chapter 4

Results

There are no secrets to success. It is the result of preparation, hard work, and learning from failure.

Colin Powell

As explained in Section 3.4 and subsequent subsections, the algorithm is a key part of the service that is responsible for updating the *computedPriority* of users. The simulations and subsequent results will be presented in this chapter. Moreover, the optimizers are crucial components of the system and will be evaluated in this chapter. Furthermore, the results regarding the database locking issues will be presented.

4.1 Evaluating the algorithms

Evaluation of the algorithm is a process consisting of two parts. The initial evaluation is done through simulations in an isolated environment. The second part is to evaluate the algorithm in production and monitor the results to verify that the resulting values are reasonable and that the system is working as expected.

4.1.1 Simulations

A set of simulations were created to be able to evaluate the results of the algorithm without running the compiled JAliEn binary on the local machine. Instead the simulations were started in a separate temporary application in the same software project. This was to have a controlled environment where the algorithm could be tested with different parameters and configurations. I created a controlled environment within a fork of the JAliEn Git-project to be able to leverage existing code and database connections. The simulations were run on a local machine and the results were stored in CSV files. The results were then plotted using Python scripts to visualize the results. The simulations were run with different configurations to see how the algorithm would behave based on a multitude of input parameters and how this would affect the fairness. The simulation ran the algorithm on each users' input values and determined who would get the next job based on the computed priority.

Original user scheduling algorithm simulations

For the original algorithm the simulation was set up with multiple stub users created with different input parameters. Only a single job was added on each iteration of the simulation as this algorithm didn't support multiple cores. The results are visualized in Figure 4.1. The results of the simulation show that only three out of six users were allowed to start any jobs out of the initial 10,000 iterations. This indicates that three of the users were not able to receive any CPU-time due to their low *computedPriority*. This is a strong indicator that the fairness of the algorithm is not optimal. From the plot it is clear that users 1, 3 and 4 are the only ones that are able to start jobs. Refer to Table 4.1 (p. 82) to observe that these users are the ones with the highest baseline priority. User 3 has the highest base priority and runs close to 2,000 jobs before its *computedPriority* is lower than the same value of user 1. After user 1 begins to receive CPU-time the plot shows that the two users are trading first place in who has the highest *computedPriority* until user 4 manages to start its first job when the total core usage is nearing 6,000. From this point on the users are trading places in who has the highest *computedPriority* until the simulation ends at 10,000 jobs, which is equivalent to CPU cores in use in this scenario. The result from simulations running the original algorithm shows that the fairness is not achieved and this indicates

that there will be cases where users are not able to start any jobs. This correlates with the information given by the domain experts on the weakness of the algorithm and presents a major issue in JAliEn that needed to be solved.

All subsequent simulations used seeded random number generation to decide how many jobs should be started on each iteration, this was to be able to recreate the exact results with the same input parameters. This method was written specifically for these simulations.

Improved fairness with multi core support

The simulations conducted using the new algorithm were similar to those discussed in the previous paragraph. What is different from the previous simulation is that this algorithm supports multi core jobs. Furthermore, the simulations on the improved algorithm utilizes a method explained in Section 3.6.1 (p. 72) to provide a seeded random number of cores for each job that should start, specifically for each iteration of the simulation. The results of the simulations are visualized in Figure 4.2. The results demonstrate that all users are able to initiate jobs, indicating a significant improvement in the fairness of the algorithm. As illustrated by the plot, each user successfully initiates their first job early on in the simulation. The plot line exhibit a competitive linear growth pattern, wherein users compete for resources and maintain the ability to start jobs from the beginning to the end. Consistent with the previous simulation, having a higher base priority is advantageous, enabling them to start a greater number of jobs. The improved fairness is achieved by progressively penalizing resource usage over time. Users with higher base priority will still have an advantage, but if they use too many resources, while beneath quota, the punishment will be severe. Although fairness is significantly improved, the simulations and testing indicated that a punishment so severe for resources consumed would not be sustainable in production, due to the large number of batch jobs submitted by the most important users. This indicated that basing a new and fairer user scheduling algorithm on the legacy algorithm would not be feasible.

Simulations on the novel algorithm

Similar to previous simulations of the legacy and the initial approach to an improved algorithm, the subsequent approach was tested in a controlled

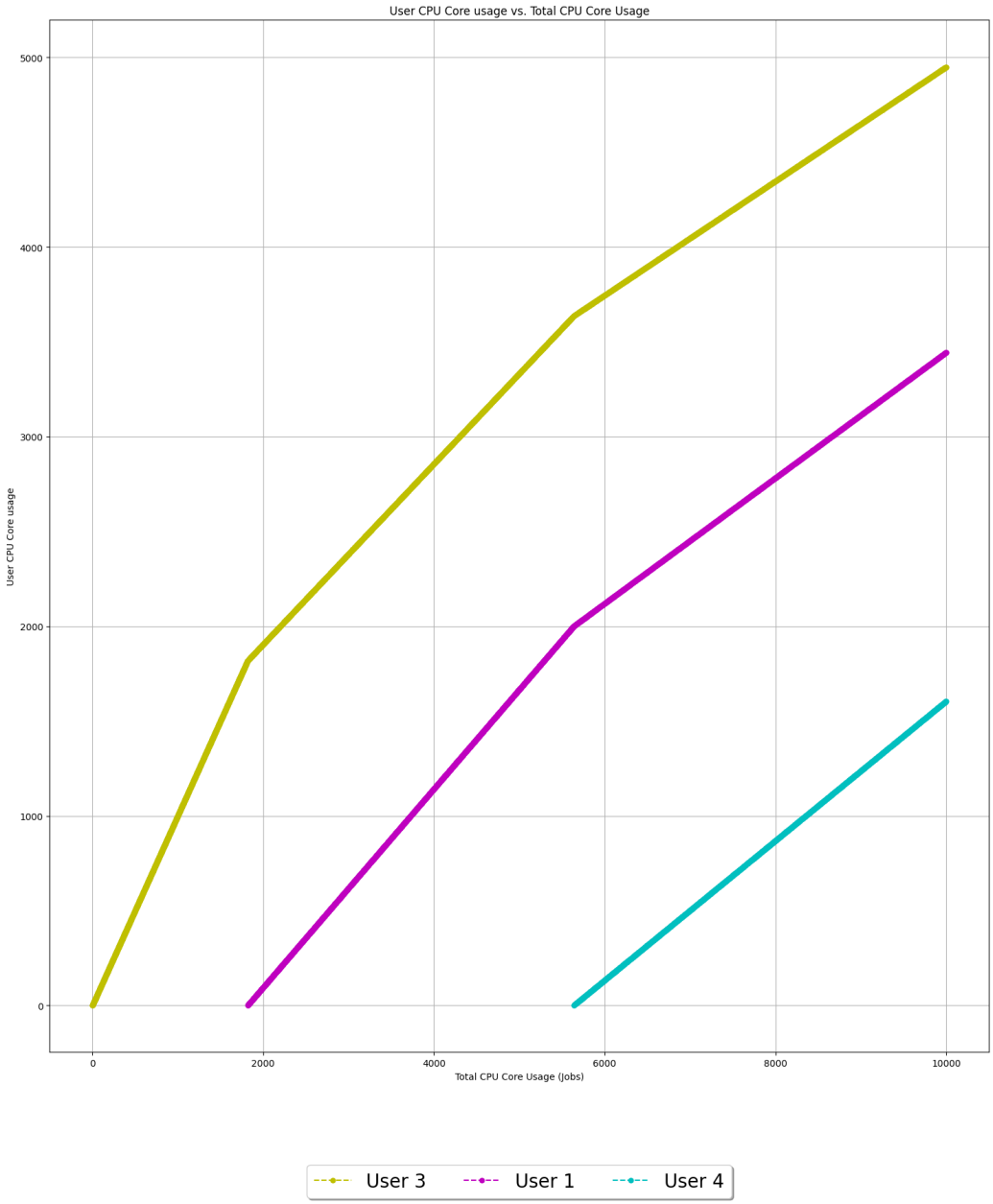

Figure 4.1: Simulation plot running 10,000 jobs with the legacy algorithm where only single core jobs are supported. The users for the simulation are displayed in Table 4.1 on page 82. Only the three users with the highest base priority are able to start jobs.

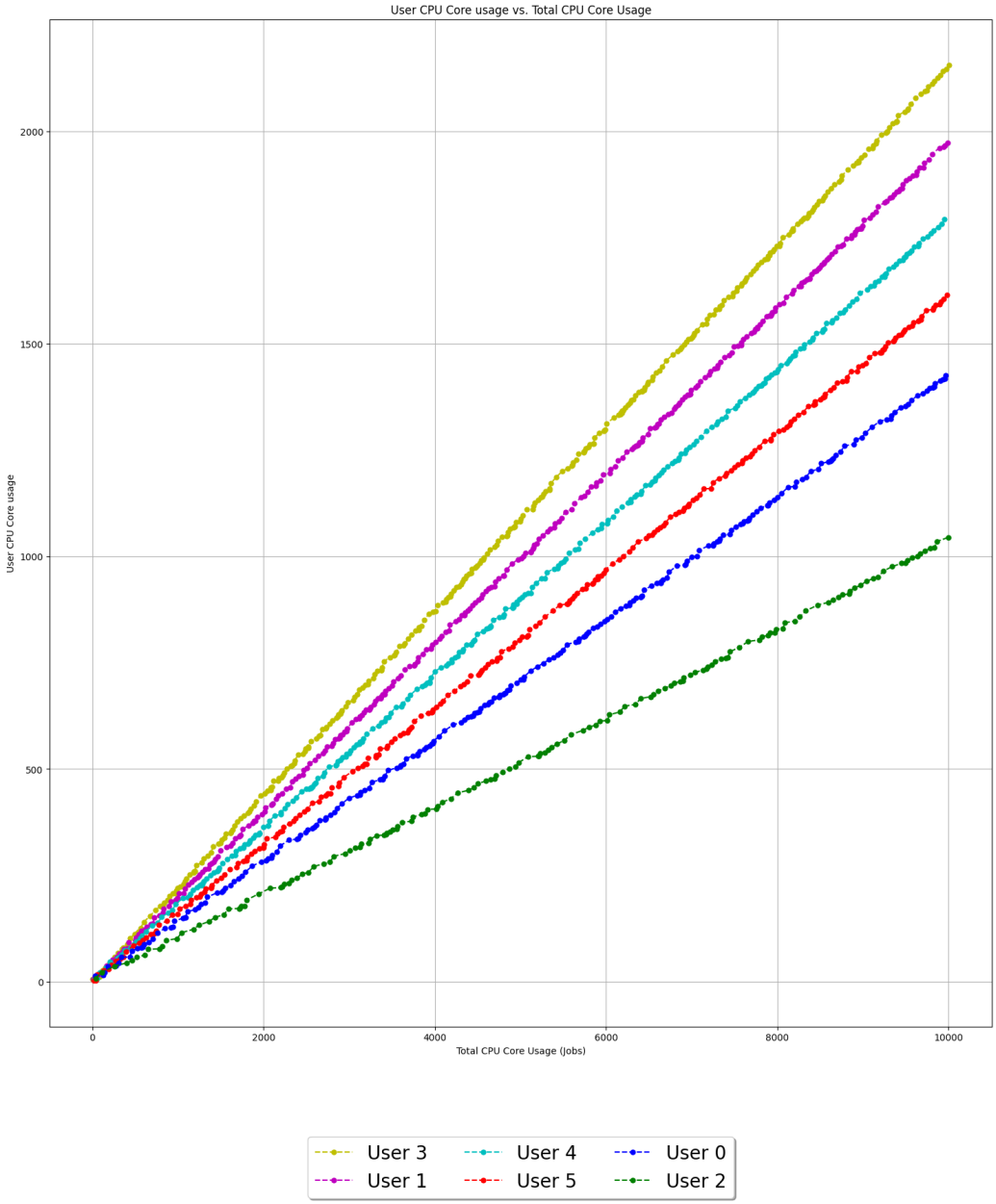

Figure 4.2: Simulation plot using 10,000 CPU cores with the updated algorithm that supports multi core jobs. Initially and through to the end all users are able to obtain CPU cores. The range between user 3 and user 2 is larger than in Figure 4.3 that uses the novel user scheduling algorithm. The users for the simulation are displayed in Table 4.1.

environment. The details of this approach to the new algorithm is explained in Section 3.4.2. The simulations were based upon the previous simulations, however it was rewritten to better incorporate more input parameters and use both production data and dummy data as input. Additionally, a more significant set of simulations were executed and a larger suite of plots followed these results. It was crucial to test the baseline priority of users, the number of cores in use and the resource cost with a variety of initial values. The baseline priority is referred to as *priority* in code examples and input data for simulations, found in Table 4.1 (p. 82). The algorithm can be seen in Listing 3.4, while the full code is available on Gitlab, see Appendix A.

Simulations using this algorithm was performed on two sets of users, one set of fictional users where they are identified from 0 to 5 and second set is based on real users. The users from the first set are identical to those in the previous simulations mentioned in Section 4.1.1 on page 77 and 78

The stub users were simulated with up to a 100,000 CPU cores in use, but for visibility the following plots displayed in Figures 4.3, 4.4 and 4.5 show the initial distribution of 10,000, 1,000 and 100 CPU cores respectively. The plots show that all users are able to obtain CPU cores and that the users with the highest baseline priority obtains the most cores. The range of cores between the user with the highest and lowest baseline priority is smaller than the previous simulation in Figure 4.2. The x-axes of the plots show the total CPU core usage, while the y-axes shows the amount of CPU cores used per user. The input parameters for these simulations are displayed in Table 4.1. The simulation results demonstrate a significant improvement in the fairness of the algorithm over time, compared to the original user scheduling algorithm.

The simulations performed on the real users were run for 100,000 iterations and the user input data was clean and prefilled. In some simulations the resource costs were set to 0, while others used real data directly from the production database as a starting point. This was to investigate if the algorithm behaves similarly in several cases. Furthermore, the simulation executed on multiple different baseline priorities in an interval that mimics the default priority of users as Aliprod, Alidaq, Alihyperloop and Alitrain. These users were chosen as they are the most important users in JAliEn, they are the 'production users', these are automated users who submit significant amounts of physics data to be processed on the Grid. It is critical that the

novel algorithm works well for these users. Therefore, these four users and one low priority user were chosen as the test data for the simulations. Table 4.3 (p. 88) shows partial results from the simulations for these five users where two of the production users had a baseline priority of 12,000. One had a baseline of 20,000, the remaining users had a baseline of 1 and 10. The first five lines show that several users have received a boost, explained in Section 3.4.2, that ensures that user 1235997 with a baseline priority of 1 were able to start a job. The entire simulation is plotted in Figure 4.6 and a subset of these results are plotted in Figure 4.7. This plot indicates that all users are allocated CPU time due to receiving a boost. The user with the highest priority starts multiple jobs. Table 4.4 on page 88 shows the last five entries of the simulation in which three users have used the majority of the CPU cores. The last production user was not starting any jobs after the first ones due to a lower *computedPriority* as it starts with a significantly lower baseline priority than the others, as seen in Table B.2 (page 110).

The plot in Figure 4.6 indicates that the user with the highest baseline priority is able to initiate a significant amount of jobs after the initial boosted values, which are not seen in the plot due to the magnitude of data points, before the next user. The input values are displayed in Table B.2. As the CPU core usage approaches but remains just below 20,000, all three users are able to initiate new batch jobs. Neither of the users reached their maximum for active cores in this scenario. The input parameters for these simulations can be found in the appendix, see Appendix B.

	User 0	**User 1**	**User 2**	**User 3**	**User 4**	**User 5**
Baseline priority	700.0	1000.0	500.0	1100.0	900.0	800.0
Max Parallel Jobs	20000	20000	20000	20000	20000	20000
Max Total CPU Cost	1×10^{14}	1×10^{14}	1×10^{14}	1×10^{14}	1×10^{14}	1×10^{14}
Total CPU Cost Last 24h	0.0	0.0	0.0	0.0	0.0	0.0
Active CPU Cores in Use	0	0	0	0	0	0

Table 4.1: Initial stub user input parameters for simulations used for all algorithms discussed in Section 4.1. Furthermore, the simulations with with these users are displayed Figures 4.1, 4.2, 4.3, 4.4 and 4.5.

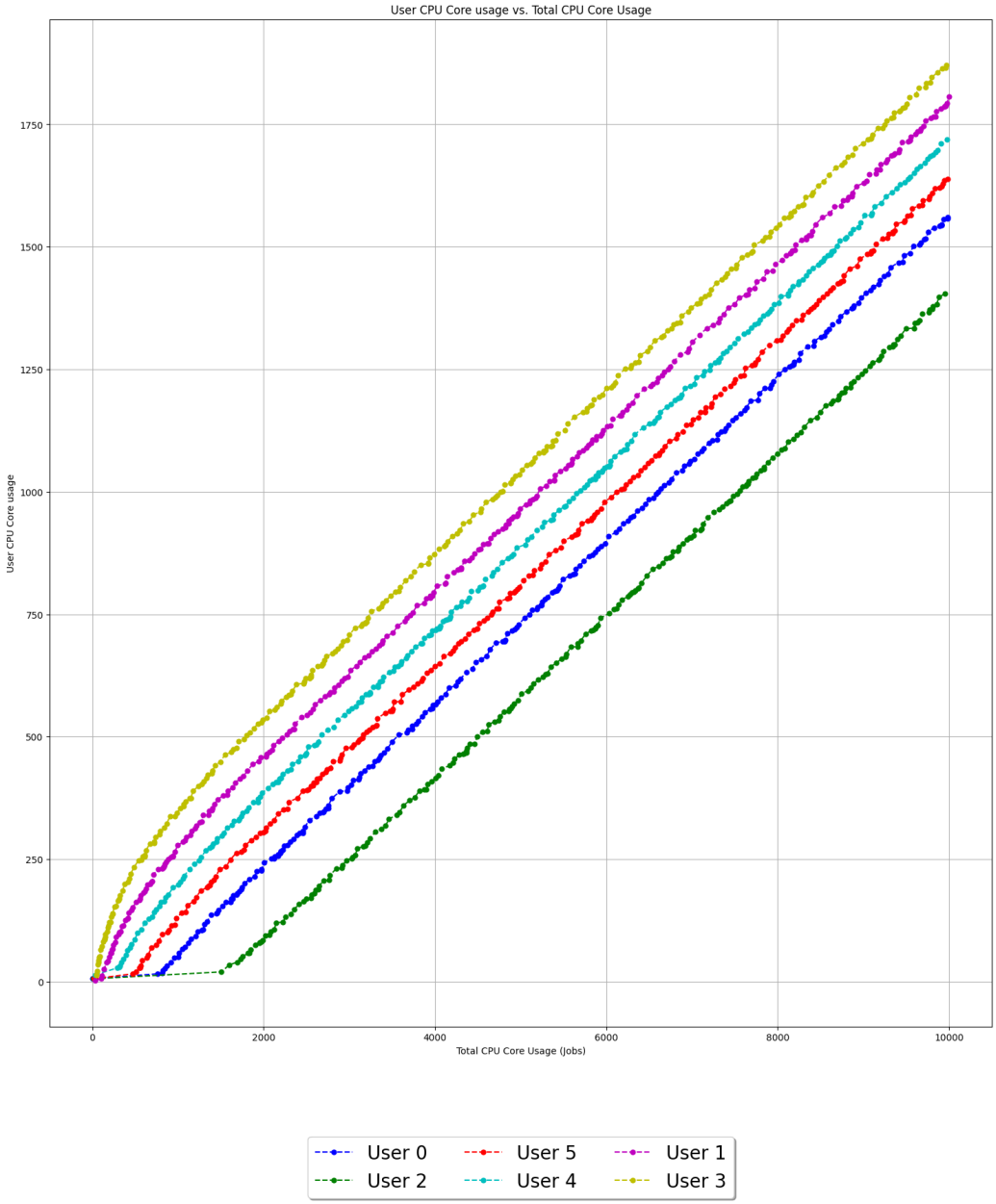

Figure 4.3: This plot represents the distribution of the initial 10,000 CPU cores among six stub users. The users and their base values are listed in Table 4.1. The plot shows a near-linear growth in core allocation, with all users eventually utilizing CPU resources. User 2, having the lowest baseline priority, had to wait until more than 1,500 cores were in use before consistently receiving cores. A granular view of the initial allocation boost is provided in Figures 4.4 and 4.5.

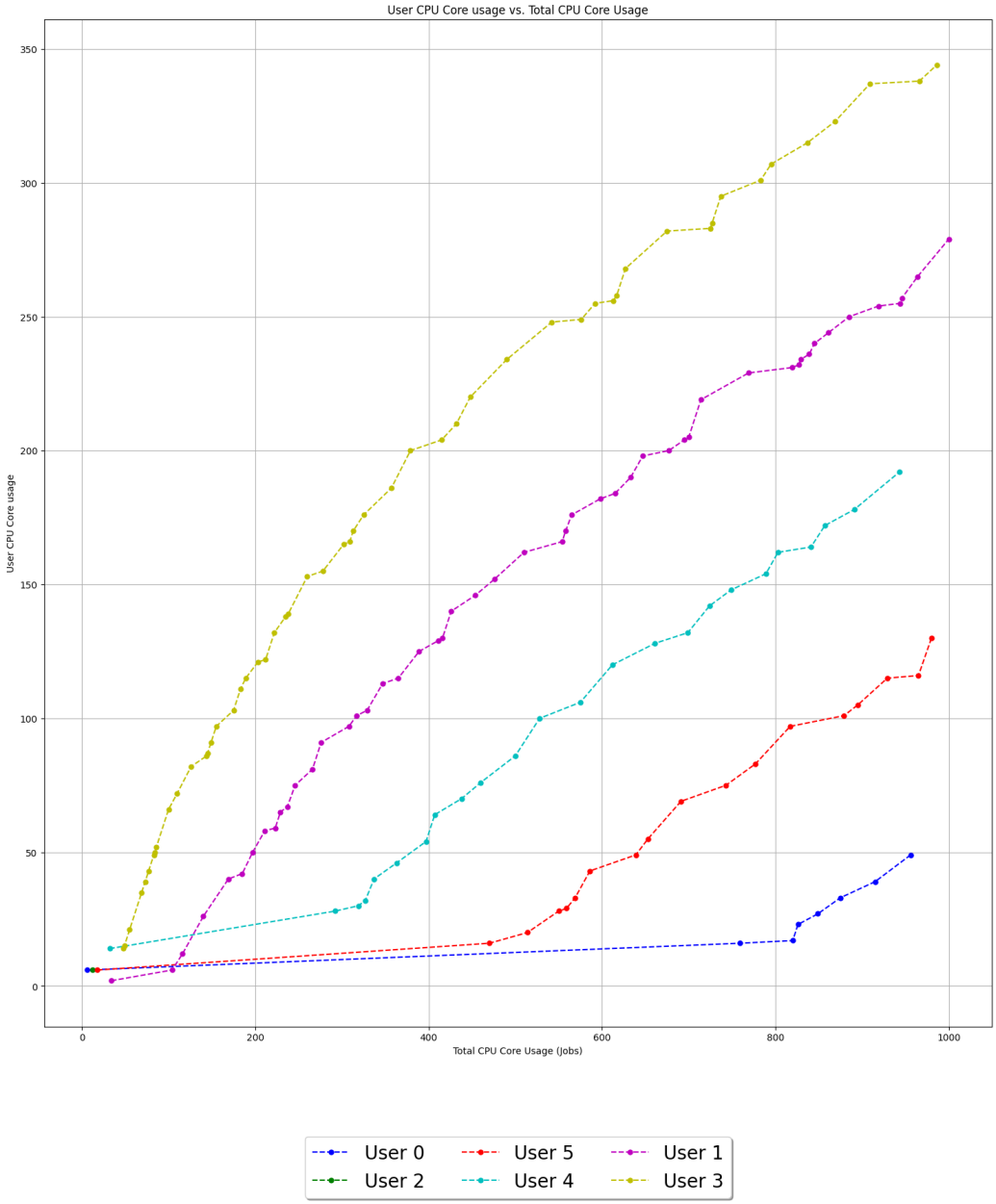

Figure 4.4: This plot represents the distribution of the initial 1000 CPU cores among six stub users. Initially all users receive a boost and obtains CPU cores, subsequently the plot shows that multiple users have to wait different lengths in time before their next core is assigned. This is the same dataset as Figure 4.3.

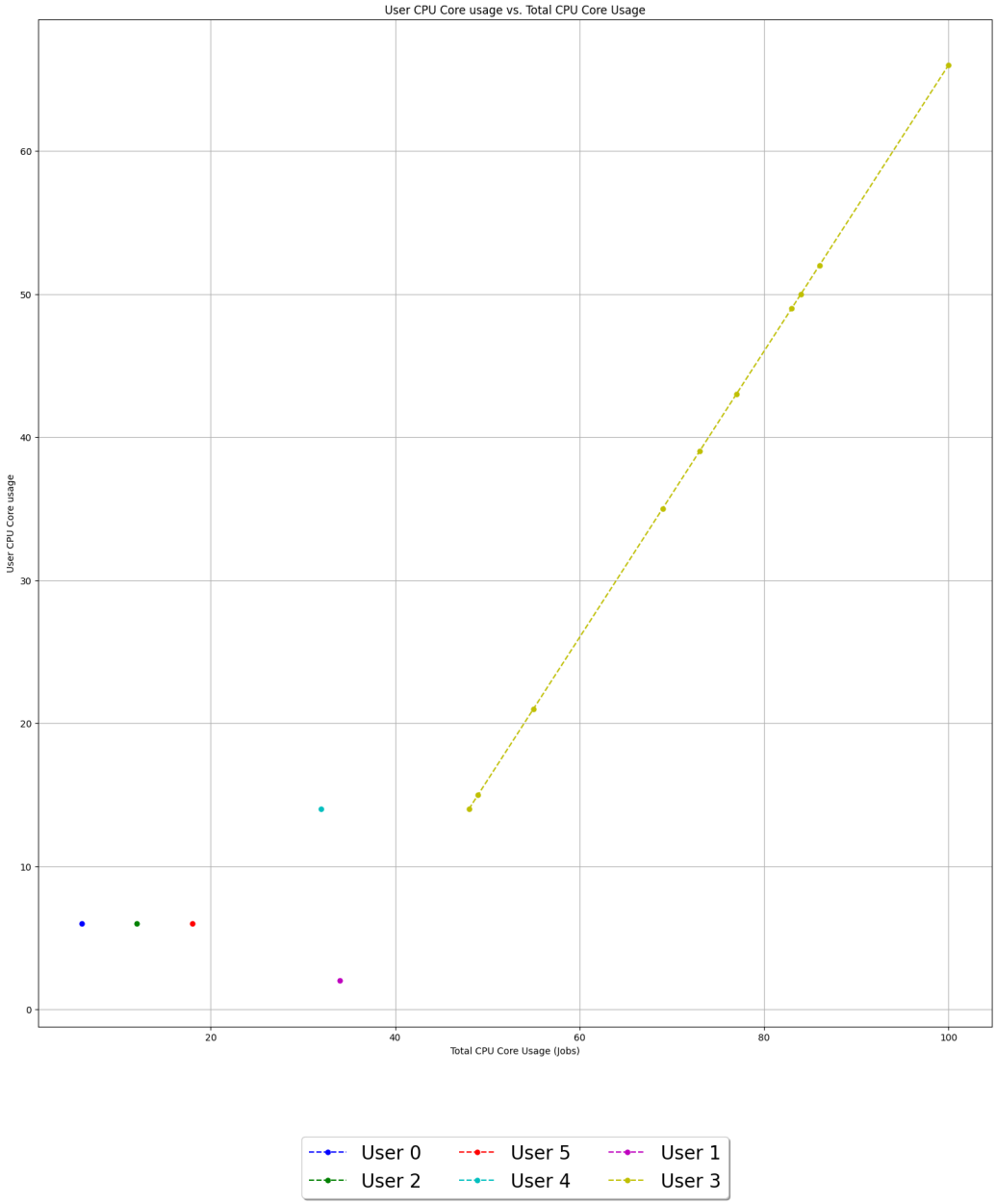

Figure 4.5: This plot represents the distribution of the initial 100 CPU cores among six stub users. Initially all users receive a boost and receives CPU cores as indicated by the single dot on the plot. This plot proves that every user is allocated cpu cores initially, before user 3 with the highest base priority receives consecutive cores. This is the same dataset as Figure 4.3.

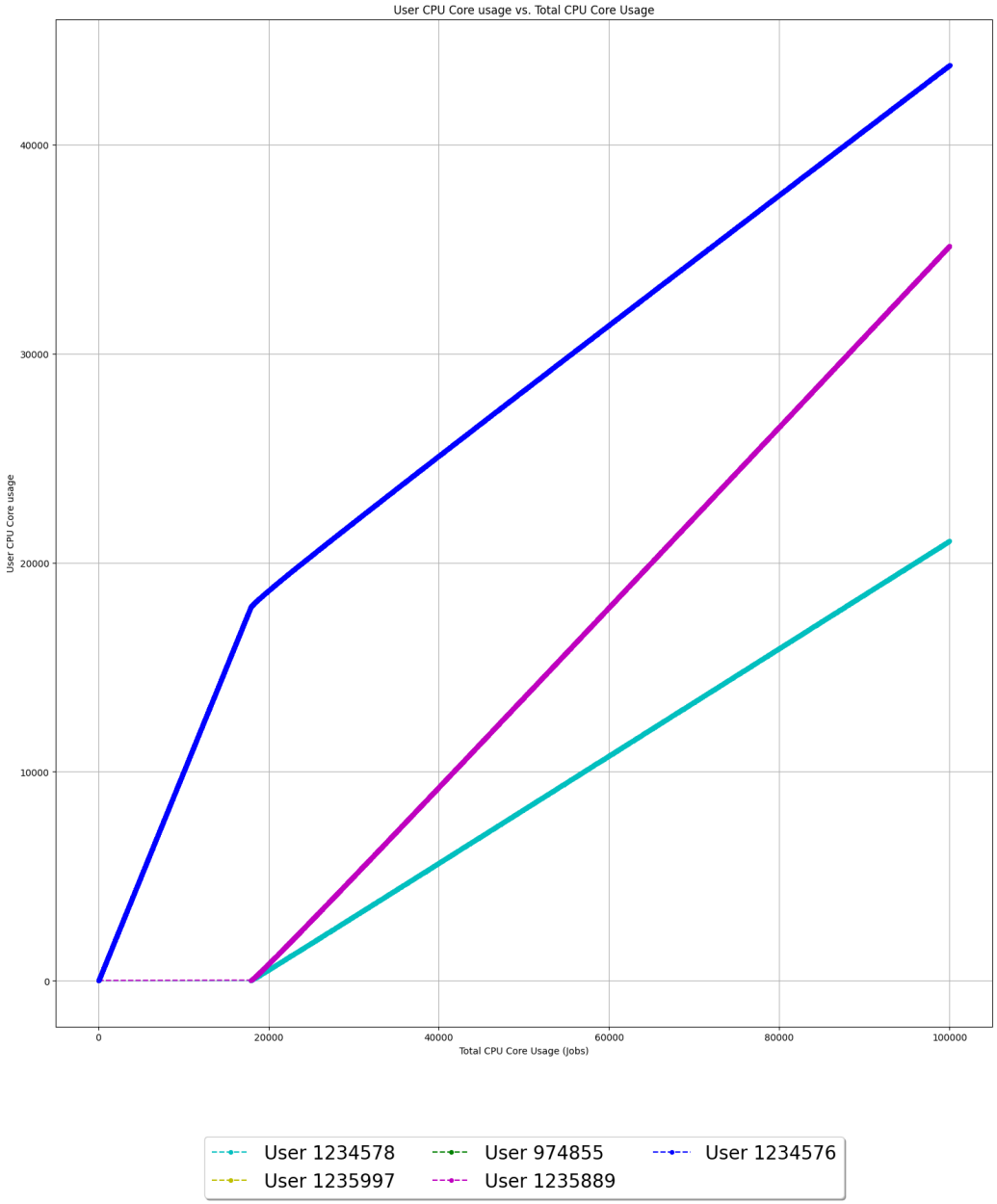

Figure 4.6: Displays three lines to show CPU core usage by production users where the largest baseline priority is 20000 and starts a significant amount of jobs before the other two are users with baseline 12000. User 1234578 utilizes fewer cores because the *computedPriority* is lower over time due to different input parameters, specifically the 35000 in difference in maximum jobs to run in parallel, as seen in Table B.2 (p. 110).

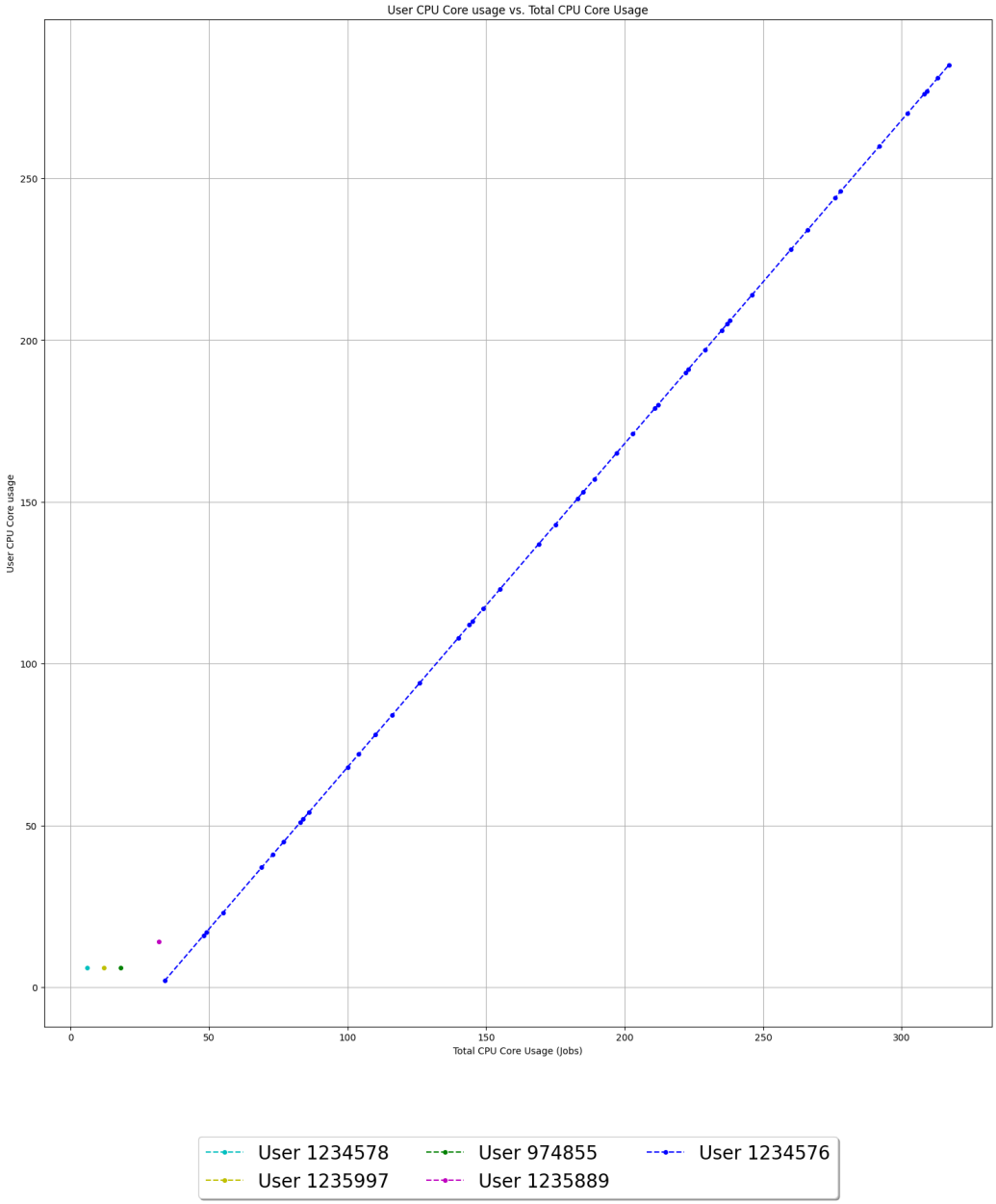

Figure 4.7: This figure shows the same dataset as in Figure 4.6, focusing on the first 300 cores consumed by the users. All users receive CPU cores, with user *1234576*, who has a baseline priority of 20,000, obtaining significantly more cores. The single dot of each color represents the initial boost allocated to the users. The initial input parameters are detailed in Table B.2 (p. 110). The full overview of the input values can be found in the appendix, see Appendix B in Table B.2 (p. 110). 87

	User 974855	U: 1234576	U: 1234578	U: 1235889	U: 1235997
Baseline priority	10.0	20000.0	12000.0	12000.0	1.0
Max Parallel Jobs	200000	60000	50000	85000	1000
Max Total CPU Cost	1×10^{12}	1×10^{12}	1×10^{12}	1×10^{14}	1×10^{7}
Total CPU Cost Last 24h	0.0	0.0	0.0	0.0	0.0
Active CPU Cores in Use	0	0	0	0	0

Table 4.2: Initial user input parameters for simulations of production users used to evaluate the novel algorithm. The plots from the simulations using these users are displayed in Figures 4.6 and 4.7.

Total CPU core usage (Jobs)	User ID	User CPU Core usage	Computed Priority start of Job	userCurrentCost
6	1234578	6	0.9900000095367432	4.32E8
12	1235997	6	9.399979591369629	4.32E8
18	974855	6	9.399800300598145	4.32E8
32	1235889	14	9.15999984741211	1.008E9
34	1234576	2	9.0	1.44E8
48	1234576	16	0.99986332654953	1.152E9
49	1234576	17	0.9995272755622864	1.224E9
55	1234576	23	0.9995085000991821	1.656E9
69	1234576	37	0.999401390552520 8	2.664E9
73	1234576	41	0.9991755485534668	2.952E9
77	1234576	45	0.999114990234375	3.24E9
83	1234576	51	0.999055802822113	3.672E9
84	1234576	52	0.9989690184593201	3.744E9
86	1234576	54	0.9989547729492188	3.888E9
100	1234576	68	0.9989264607429504	4.896E9
104	1234576	72	0.998733639717102	5.184E9
110	1234576	78	0.9986799955368042	5.616E9
116	1234576	84	0.9986006021499634	6.048E9
126	1234576	94	0.9985222816467285	6.768E9

Table 4.3: This table presents the results of initial simulations on production users. It demonstrates that user *1234576*, with a higher priority, acquires multiple CPU cores after the initial distribution. The input parameters for these users are detailed in Table 4.2 (p. 88) and corresponding plots are shown in Figures 4.6 and 4.7.

Total CPU core usage (Jobs)	User ID	User CPU Core usage	Computed Priority start of Job	userCurrentCost
99981	1234578	21038	0.6173743009567261	1.5147439E12
99987	1234576	43791	0.617369532585144	3.152988E12
99989	1235889	35148	0.617351233959198	2.530666E12
99995	1235889	35154	0.617339015007019	2.531098E12
100009	1234576	43805	0.617318332195282	3.153996E12

Table 4.4: This table displays the final state of the simulations on production users. It shows that user *1234576*, with the highest priority, retains a significant number of CPU cores by the end of the allocation process. The input parameters for these users are provided in Table 4.2 (p. 88) and the related plots are depicted in Figures 4.6 and 4.7.

4.2 Evaluating the optimizers

Each optimizer operates independently, even though some, like the *PriorityRapidUpdater* and the *PriorityReconciliationService*, are closely related and require similar data inputs. Despite their interrelated functions, they can still run autonomously. Therefore, the optimizers were tested individually to verify the functionality. Several tests were conducted where multiple optimizers were executed sequentially to observe the results and to observe if optimizers who interact on the same table caused processes to queue up due to long locking times. The insights gained from over six months of running the optimizers in production suggest that they do not cause many processes to be queued when interacting with the same database. Similarly to the simulations of the algorithms, the optimizers were tested in isolated environments on a local machine. When the testing satisfied the criteria of the optimizer they were deployed and tested in the development environment and in the production environment at a later time. Before initiating local tests, a setup configuration was executed to provide the necessary test data for simulating the optimizer in action. The optimizer was instantiated and started independently of the JAliEn application. This allowed observation of the optimizer's behaviour and verify that the optimizer was functioning as expected. All optimizers interact with the central job related database. For testing, a JAliEn development database was used, featuring replica users that closely resemble the production users. This facilitated testing of an individual optimizer on its own, or in conjunction with other optimizers. The output from the optimizers is written to standard out and to a persisted file to provide a log of the optimizer's actions. This log was useful for debugging and verifying that the optimizer was functioning as expected when integrated in JAliEn and deployed production.

Specifically the evaluation of the *OverwaitingJobHandler* optimizer from Section 3.5.8 required a distinct testing approach. Verifying the behaviour through investigating logs and the database tables confirmed the functionality of the optimizer. However, testing the edge case in where the worker node is disconnected and determining whether the running batch job would recover after the connection was reestablished after an hour, required a meticulous strategy. This was achieved by starting JAliEn locally to act as a worker node and submitting a batch job to observe the behaviour. The job was configured to sleep for over an hour before completing. Meanwhile, the machine

was disconnected from the internet and reconnected after the hour passed, but before the job completed. The expected behaviour was for the batch job to be set to the ZOMBIE state to indicate that no heartbeat was received from the worker node. The result was as expected and the worker node was able to recover and continue the job after the connection was reestablished. This test was important to verify that the optimizer would be robust enough to handle such scenarios in production.

4.3 Problems with delayed database access

The issue of many processes queuing up in the database posed a significant problem for the system, causing delays and inefficiencies, and served as a major motivation for the challenges addressed in this project. As outlined in Section 1.2, the continued operation of the bookkeeping service in production was the main source of the problems. Resolving the issues was not just a matter of implementing a new service; it required a comprehensive understanding of the service's functionality and its interaction with the database. Multiple optimizers contribute to solving the problem, with the two primary ones being the *PriorityReconciliationService* and the *SitequeueReconciler*.

The general approach to resolving the issues was to break larger queries into smaller ones, using dirty read, updating row by row and to avoid complex queries that combine read and write operations by writing explicit queries for each task. Reducing the number of locks taken for all SELECT operations, by utilizing dirty read, was helpful to maximize the time when locks were available for other processes. This approach made a significant difference by reducing the number of processes waiting in queue to access the database compared to the original method, which involved calculating the information to be updated and updating all rows in a single transaction. This is explained in detail later in this section. While the total number of processes accessing the database remained the same, fewer processes were left waiting, thereby decreasing wait times and improving overall efficiency. Moreover, while executing queries row by row instead of in batches may increase the total execution time, this trade-off is advantageous for JAliEn. As previously outlined in Sections 3.4 and 3.4.2, the original user scheduling algorithm was embedded within a database update query. By extracting this functionality and dividing the update process among multiple optimizers, the strain

on the database was significantly reduced. The optimizers *ActiveUserReconciler* and *PriorityReconciliationService*, which are responsible for updating the *computedPriority* of users, both leverage the aforementioned approach by separating the concerns of calculation and update into distinct processes.

There are two optimizers that became the primary solutions to the issues where thousands of processes were queuing up when attempting to access the database. These optimizers are the *PriorityReconciliationService* and the *SitequeueReconciler* optimizer. The first optimizer is tasked with reconciling resource usage values from the QUEUE and QUEUEPROC tables and computing *computedPriority* and storing *computedPriority* in the PRIORITY table. The latter optimizer is tasked to summarize the number of jobs per state and the cost for each site. This is an intensive operation and when executed as a single query on the entire SITEQUEUES table, it caused many rows to be locked. When other queries executed simultaneously it could result in the database being locked for an extended period, significantly hindering and delaying recovery. To mitigate the issue momentarily, the domain experts implemented a script that would restart the database whenever the number of processes waiting to access it exceeded 4,000. This was a temporary solution to keep the system running, but it was not a sustainable solution. The *SitequeueReconciler* optimizer was developed to resolve the issue by separating the read, summarization and write operations from each other, while maintaining matching features as the original service. This meant that no read locks would be taken and the summarization of the values would happen separately from the database operations. After the values are summarized they are written row by row to the SITEQUEUE table. The outcome of deploying this optimizer was significant and clearly visible in Figure 4.8 which displays over 400 restarts of MySQL between early February to early May due the threshold of waiting processes being exceeded. Postfactum, the number of restarts was reduced to a single event, which was unrelated to the problems described above.

By counting the number of active connections to the database for three seconds, it was verified that the number of processes that queue up to access the database during regular load is approximately 1,200, or 400 processes queued per second. The query used to count the connections is displayed in Listing 4.1. The execution of the two legacy and novel queries were timed as they were the most impactful optimizers in the duration of execution time. These values are relevant to answer research question 3 which is presented in

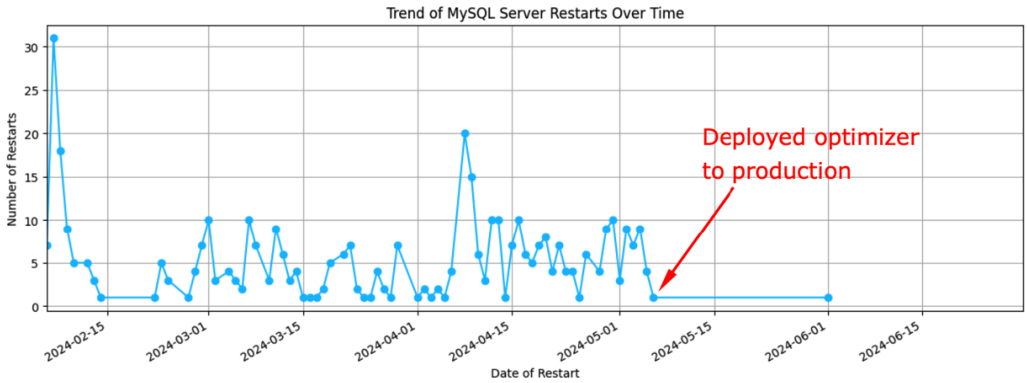

Figure 4.8: This figure shows the number of MySQL restarts over a four-month period. Between February 5th and May 6th there were 406 restarts as plotted in the graph. The graph displays the number of restarts per day, with the x-axes representing the date and the y-axes showing the number of restarts. About a month after the deployment of the *SitequeueReconciler* optimizer there was a single restart, which was an unrelated event.

Section 1.3 and asks: *How much can the total duration of JAliEn TaskQueue the database locking duration be reduced by splitting a large operation into smaller queries?*. The execution of the original query which calculated *computedPriority* and locked entire tables, took approximately 1.5 seconds to complete. This operation was executed on approximately 3,000 users this gives about 0.5 milliseconds locking duration per user.

$$\text{Lock time per row} = \frac{1500\,\text{ms}}{3000\,\text{rows}} = 0.5\,\text{ms/row} \tag{4.1}$$

```
while true; do ./mysqlconsole.sh 3308 -e "show
   full processlist;" 2>/dev/null | grep -v
   Sleep | wc -l; sleep 0.1; done
```

Listing 4.1: Counting the number of active connections to the database over a three-second period.

This operation is now rectified in the *PriorityReconciliationService*, that implements the novel user scheduling algorithm, resulting in minimal locking durations. By only updating row by row for users that had any changes in their resource usage the number of affected rows is reduced. Furthermore,

this approach allows other processes to obtain read and write locks in between each write lock. The execution time of the entire optimizer is measured to be longer than the original, however the speed of the entire operation is not the most important thing as the reliability of the database is more important. As noted in earlier sections, it is worth trading longer execution time for shorter locking durations. The optimizers' execution time was measured to a total of 9.5 seconds. The majority of the duration, 9.3 seconds, was to retrieve the users without locking and calculate the new values to be updated. The update operation took ≈ 0.26 milliseconds per user, or ≈ 780 ms for all existing users. This operation requires more overall clock time to complete, but it allows other processes to access the database between each row update.

$$\text{Lock time per row} = \frac{780\,\text{ms}}{3000\,\text{rows}} = 0.26\,\text{ms/row} \tag{4.2}$$

The legacy operation that updated the SITEQUEUES table would summarize the states for each site and count the cost per site. The execution time of this operation was measured and the total duration of this query was 11.6 seconds. The entirety of this duration is seen as a period where all the required rows are locked. This would cause a significant number of processes to queue up while waiting to access the database. The variable QP represents the number of queued processes and is calculated as follows:

$$QP = 400\,\text{processes/second} \times 11.6\,\text{seconds} = 4640\,\text{queued processes} \tag{4.3}$$

The ameliorated version of this query is the *SitequeueReconciler* optimizer, that locks minimally while updating row by row. The execution time of this optimizer was measured to be 4.3 seconds. The majority of the duration, is the read and summarization of the values to be updated. The update operation finished in less than 102.3 milliseconds when updating the 610 count and cost values. As noted before, although this operation is significantly faster in clock time to execute, this is not the most prominent factor. The most important aspect is the ability for other processes to access the database while the optimizer is running.

$$OLD = 1.5\,\text{s} + 11.6\,\text{s} = 13.1\,\text{s original locking duration} \tag{4.4}$$

$$NLD = 780\,\text{ms} + 102.3\,\text{ms} = 882.3\,\text{ms novel locking duration} \tag{4.5}$$

Considering $RQ3$ the total locking duration has been reduced from the original total locking duration of 13.1 seconds to 882.3 milliseconds. The equation for this can be seen above for the variable OLD and NLD for the legacy and novel approaches. The longest locking interval in the legacy system was 11.6 seconds and less than 800 ms with the novel algorithm and optimizers. The comparable runtime of the two queries is 13.1 seconds, while the new optimizers requires 13.8 seconds to execute. An answer to the $RQ3$ is that this novel approach has reduced the total locking duration $\approx$ 12.22 seconds, as displayed in variable RD in Equation 4.6. Furthermore, it must be noted that these two queries are parts of a larger system and while they might not always align simultaneously and block for a combined 13.1 seconds, other queries might also interfere. Regardless of the specific situation, the combined locking time would lead to an immediate increase in the number of queued processes.

$$RD = 13.1\,\text{s} - 882.3\,\text{ms} = 13.1\,\text{s} - 0.8823\,\text{s} \approx 12.22\,\text{s} \tag{4.6}$$

4.4 The enhancements to the system

Solving the database queuing issues as described in the previous section was a part of the enhancements that was required in the project. Additionally, the user scheduling algorithm needed to be enhanced to improve the fairness for all users to ensure that all users are able to submit and execute batch jobs every day. The approach and the results of improving the fairness of the algorithm and the outcome have been detailed in Sections 3.4.2 and 4.1.1. As reflected in the figures above, the fairness of the algorithm has been significantly improved, ensuring that all users receive some CPU cores to execute their batch jobs. The simulation results show that a user with a low baseline priority will only receive a minor amount of CPU cores initially, over time the number of CPU cores are distributed to all requesting users.

In the production environment it was observed that the users with low baseline priority could potentially start multiple jobs and reach their maximum

number of active CPU cores and potentially exceed the threshold, by submitting and starting their batch jobs within the time interval where the *computedPriority* is recalculated. This is a caveat of the system and together with the domain experts, it was decided to reduce the time interval for the *JobAgentUpdater* and the *ActiveUserReconciler* down from five minutes to thirty seconds to prevent users from exceeding their quotas.

Chapter 5

Discussion

The good thing about science is that it's true whether or not you believe in it.

Neil deGrasse Tyson

This chapter will discuss the results from the previous chapter. Furthermore, the solution and major findings are considered in addition to section on how this approach could be used in other systems outside of CERN.

5.1 Major findings

The prominent issue to investigate was the fairness of the user scheduling algorithm from AliEn. **Research question 1** (RQ 1) was concerned with discovering *What parameters are required in a user scheduling algorithm to ensure fairness and minimal impact on the performance of the system?*. The parameters were extensively discussed in Section 3.4.2 and the results of the algorithm are presented in Section 4.1.1 (p. 78). To promote fairness, the novel algorithm was designed to prioritize users who have not consumed any resources in the past 24 hours. To achieve this a set of relevant parameters were identified, these are *cost*, *activeCpuCores* and *priority*. These chosen parameters ensure that the algorithm accounts for the user's resource consumption, the number of active CPU cores and the user's baseline priority. By using the two parameters, *cost* and *activeCpuCores* that are tied to the

user's resource usage, the algorithm can reward or penalize based on recent activity. It should be noted that these are not the only viable parameters that are available for the algorithm. An additional feasible parameter is *totalRunningTimeLast24h*. This could be in addition to or potentially instead of *cost*. As explained in Section 3.4.2 the *cost* parameter is the product of elapsed real time, the number of CPU cores utilized and the price of the job. Since the *cost* parameter already accounts for time spent utilizing resources, it could be argued that using both *cost* ($(cores * walltime) * price$) and *totalRunningTimeLast24h* (CPU time spent in the last 24 hours) would be redundant. Either parameter could be used to achieve enhanced fairness in the user scheduling algorithm compared to the legacy algorithm.

The second part of **RQ 1** focuses on reducing the impact on the performance of the system when utilizing the novel user scheduling algorithm. The solution is multifaceted, as it involves not only the algorithm itself but also the optimizers and the frequency of their execution. The two optimizers that invoke the novel user scheduling algorithm are *PriorityReconciliationService* and *ActiveUserReconciler*. As previously explained the *ActiveUserReconciler* is executed more frequently and only updating the users that used CPU resources in the previous 24 hours. The *PriorityReconciliationService* is tasked to update the *computedPriority* for all users. This optimizer runs less frequently and is designed to skip updating users whose values have not changed. This approach reduces the number of rows that need to be locked for updates, thereby minimizing the impact on system performance.

The next research question, **RQ 2**, was concerned with investigating *What is an optimized approach to sum and update user values, while minimizing system impact and having a short database lock duration?*. The proposed solution focused on minimizing impact on system performance through optimizing the database queries by splitting complex queries into smaller simpler ones. Additionally, the calculation and summarization of values are extracted from the database queries of AliEn and moved to the code. As explained in the previous sections, the database lock duration is minimized by utilizing dirty read for all select statements, significantly reducing the number and duration of locks required on rows in the tables. Moreover, this limits the required database locks to only the write operations that must be executed. This approach ensures that the system can continue to operate efficiently even when the database is under heavy load and many queries need to be executed.

The prominent complex queries are the *SitequeueReconciler* and *PriorityReconciliationService* that are discussed in Section 4.3. Furthermore, there are other examples of queries that are optimized in the code or by utilizing the database. The *JobAgentUpdater* is an example of a query that is optimized by utilizing the database to transfer information about *computedPriority* between the PRIORITY and the JOBAGENT tables. This approach is efficient and quick as it only requires a single query to update the values in the database. The simple update query displayed in Listing 5.1 can complete in under 3 milliseconds and has a low impact on the system. The functionality provided by the *JobAgentUpdater* optimizer is essential for the matchmaking capabilities of JAliEn to match a job to a site, as the *computedPriority* value is required in the matchmaking process.

There are cases where letting the database handle the workload is advantageous, but in some instances, it is more efficient to perform the heavy processing within the code. A different strategy is leveraged for the *MasterSubJobReconciler* where there are two select and update queries, to find master jobs in running and in final states. These are divided into two select queries, allowing for non-locking reads and simplifying the update process, as they handle either completed or still-running jobs. The functionality of the *MasterSubJobReconciler* optimizer is explained in Section 3.5.10 (p. 71) and the code for this and other optimizers can be found in Appendix A (p. 106). The strategies described above explains the optimized approach to sum and update user values, while minimizing system impact and reducing database lock duration.

```
1 UPDATE JOBAGENT INNER JOIN PRIORITY USING(userId)
    SET JOBAGENT.priority = PRIORITY.computedPriority
```

Listing 5.1: Updating the *priority* column in the JOBAGENT table for all users based on the values in the PRIORITY table.

The results of the final research question, **RQ 3**, are extensively presented in Section 4.3, explaining the prominent cause of the long locking time problems and the solutions. Additionally, the question of *How much can the total duration of the JAliEn TaskQueue database locking duration be reduced by splitting a large operation into smaller queries?*, indicating that the total locking duration was reduced by approximately 12.2 seconds for the two most complex and time-consuming queries combined. The timings obtained

are a combination of manual database queries and optimizer executions. The timing results from the legacy system are based on manual queries, as the AliEn optimizers are no longer operational. In contrast, the timings for all new optimizers are captured during their execution and automatically logged to a persistent log file. As both sets of queries are performed on the same production database the results are comparable. Furthermore, as explained in Section 3.4.2 (p. 57), the novel user scheduling algorithm leverages a weighted sum (similar to linear interpolation) to calculate the final priority value based on the input parameters. The frequency of the recalculations of the *computedPriority*, using the algorithm, is higher in the JAliEn now than it was in the previous system. The reduced system impact and shorter locking duration make this possible, allowing the *computedPriority* value for users to be updated more frequently, thereby enhancing the system's correctness. Additionally, this helps prevent users from exceeding their quota before their priority is updated.

Comparatively, the total execution time for the two new optimizers is shorter than the legacy optimizers and the total locking time is significantly reduced. The total locking time for the two most time-consuming queries has been reduced from over 13 seconds to less than 1 second. The new approach provides a significant improvement in speed and increases the system's effectiveness by allowing other processes to access the database between each write lock.

5.2 How can this be used in other systems

It is important to note that JAliEn presents a particular case, as it does not necessitate absolute correctness of all values at every moment. Correctness is significant, but due to the regular reconciliation of values, the correct state will be balanced regularly and the values will be up-to-date across all instances of JAliEn. Other systems, like a bank system handling money in a user account for instance, would require information to be exact at all times, and for these systems, using an approach to read uncommitted data from the database would not be suitable to ensure consistency. Moreover, there are use cases where a response that is not fully up-to-date may be perfectly acceptable, such as a Google search, where a quick but approximate answer is often preferred.

The novel user scheduling algorithm is not a scheduling algorithm in the tra-

ditional sense as introduced in Section 2.1 (p. 19), as it does not perform the actual scheduling of jobs. However, the novel algorithm shares similarities with priority scheduling algorithms, where a priority is associated with each process—in this case, each user. Instead, a single value like *computedPriority* provides a precomputed metric in the database, allowing for efficient filtering or matching of resources. The job scheduler in JAliEn requires the *computedPriority* value, calculated by the novel user scheduling algorithm, to select the job with the highest user priority for execution. The approach with eventual correctness works for JAliEn together with this novel algorithm, but there are no requirements in the algorithm that would prevent it from being used in other systems given similar parameters and users. The weights are adjustable and could be changed or new parameters could be added to the algorithm to fit the requirements of other systems. The current weights are *cost*, *activeCpuCores* and *priority* (baseline priority for the user) as displayed in Table 3.5 (p. 60). The *activeCpuCores* parameter, set at 0.5, is the highest-weighted parameter, slightly above *priority*. This indicates that the number of cores a user is utilizing is the most significant factor, though it can be adjusted up or down based on system requirements. The *priority* parameter is set to 0.4 and has a impact on the weighted sum of these weights, as the aforementioned production users in JAliEn are important and have the highest baseline priority. This parameter should have a significant weight value in this context. The final parameter is *cost* which is set to 0.1 and has a relatively small weighted contribution to the calculated value.

Testing an algorithm by running isolated simulations is a good approach to determine the feasibility of the algorithm. However, it is important to note that simulations have limitations as they do not fully represent the complexity of the environment that the algorithm exist in. In the case of JAliEn, the simulations do not take into account the matchmaking process that matches jobs for sites and the other ongoing transactions that are happening concurrently. The simulations were designed to 'start jobs' immediately after another and recalculating the *computedPriority* for all users after every iteration. In reality many jobs can be started between each recalculation of the *computedPriority* value, currently set to be executed every 30 seconds. Moreover, the JAliEn system contains other functions that are relevant when deciding which job and where to run it, as explained in Section 2.2.4 (p. 35). This is a limitation of the simulations, as they do not fully represent the com-

plexity of the system. However, the simulations are a good starting point to test the algorithm and to see how it behaves in a controlled environment. The results from the simulations can be used to determine if the algorithm is feasible and if it should be implemented in the production system. Moreover, after the simulations indicate that an algorithm is feasible, it should be tested on production data and in this case the algorithm was running on production data for months before the legacy service was replaced. The algorithm was deployed into production and has been running successfully for several months. It is an important component in JAliEn where potentially over a million jobs can be processed in a day. Furthermore, the behavior of the algorithm has been evaluated under various conditions. For instance, during slower days when the Grid was saturated with long-running jobs, priority was correctly assigned to newly submitted analysis jobs when needed. In idle conditions, as the number of jobs ramped up rapidly and required capping to adhere to users' quotas, the algorithm applied the caps as expected. Similarly, when competing users with relatively similar priorities (such as analysis jobs) shared resources, the algorithm distributed resources fairly. In all cases, the novel user scheduling algorithm performed as expected and is now running unsupervised in the production environment.

Chapter 6

Conclusion

When something is important enough, you do it even if the odds are not in your favor.

Elon Musk

As stated in Section 1.2, the work presented in this thesis aimed to improve the fairness of the user scheduling algorithm in JAliEn and to ameliorate the performance of the system by optimizing complex database queries. The research questions for this thesis were:

1. **RQ 1**: What parameters are required in a user scheduling algorithm to ensure fairness and minimal impact on the performance of the system?
2. **RQ 2**: What is an optimized approach to sum and update user values, while minimizing system impact and having a short database lock duration?
3. **RQ 3** How much can the total duration of the JAliEn TaskQueue database locking duration be reduced by splitting a large operation into smaller queries?

RQ 1 is considered answered as the algorithm's fairness is enhanced and the performance impact on the system is minimized. This is achieved through a combination of the novel user scheduling algorithm and the design of the *PriorityReconciliationService* and *ActiveUserReconciler* optimizers. These

findings are discussed in Section 5.1.

RQ 2 is answered in Section 5.1, which discusses the strategies taken to minimize locking duration and the impact on the system by breaking complex queries into smaller ones and utilizing the database for data transfer. The optimizers discussed in the aforementioned section are designed to utilize code for calculations and value summarization rather than relying on the database. This approach minimizes the number of row locks needed, thereby reducing the impact on system performance.

Finally, **RQ 3** is addressed in Section 4.3, which discusses the database issues of long locking time and their resolution. The section highlights that the total duration of the database lock was reduced by approximately 12.2 seconds for the two most time-consuming queries. The results indicate that the total locking time decreased from over 13 seconds to less than 1 second, making the process approximately 15 times faster. Additionally, this novel solution allows other processes to access the database between update queries.

In summary, the fairness of the user scheduling algorithm has been improved by implementing a novel algorithm that prioritizes users that have not consumed any resources in the last 24 hours. This approach utilizes weighted values as input parameters to the algorithm, this provides flexibility for the future if the requirements for the algorithm change. Moreover, the performance of the system has been enhanced by optimizing complex database queries. The queries were improved by breaking larger queries into smaller ones, utilizing the isolation level READ UNCOMMITED to avoid taking read locks on the database and updating row by row. The most significant changes made during the optimization process were allowing other processes to access the database between queries and updating rows individually while avoiding read locks.

The system is used by researchers all over the world to analyze data from the ALICE experiment at CERN. The results from this work have been well received by the experts at CERN and the system is now more efficient and fairer than before. The resulting code, comprising a novel user scheduling algorithm and nine optimizers, has been deployed in production within JAliEn on May 6th 2024. The code plays a crucial role in the job management system within the Grid middleware, which can process over a million batch jobs daily.

Chapter 7

Further work

The task for this project has been completed and the original bookkeeping service has been replaced. The solution is deployed in production and is used every day by users all over the world. However, there are still areas for improvement and further research and development that could be done to enhance the system.

JAliEn is a complex system that aggregates resources from over 60 computing centers worldwide. There are always new challenges that arise and complexities inherent in managing and optimizing a highly heterogeneous Grid computing system. This system, characterized by diverse CPU architectures, operating systems, batch queues, memory and I/O capabilities, demands a sophisticated approach to ensure seamless and efficient operation. The JAliEn system aims to provide a unified interface for users, but this requires continuous adaptation to the evolving technological landscape, such as changes in OS to EL9, cgroups v2 and the integration of ARM CPUs.

This future work could possibly be a PhD and focus on modeling the I/O capabilities, limits and bandwidths of storage at various sites to dynamically adjust job profiles based on site capabilities. This is crucial to prevent bottlenecks, optimize throughput and ensure stable operations across the system. By developing a dynamic, automated model that takes into account real-time site conditions, the research will enable more intelligent job scheduling, prioritizing Monte Carlo [74] simulations over I/O-intensive analysis jobs when necessary. Additionally, exploring different database and queue management

systems, such as comparing PostgreSQL and MySQL, will be part of the effort to enhance system performance. Future work like this is essential for addressing the pain points related to I/O bottlenecks and ensuring the system can adapt swiftly to hardware changes. Ultimately, this would improve the overall efficiency and reliability of the distributed computing environment by ensuring optimal resource utilization, reducing downtime and enabling the system to better handle fluctuations in workload and infrastructure.

Appendix A

Source code

The source code for the JAliEn is available on GitLab (requires CERN account):

https://gitlab.cern.ch/jalien/jalien.

The source code for the legacy AliEn is found here:

https://github.com/jaflaten/master-thesis-appendix/legacy/alien.

The source code for the algorithm and helper methods can be found here (requires CERN account):

https://gitlab.cern.ch/jalien/jalien/-/blob/master/src/main/java/alien/priority/CalculateComputedPriority.java?ref_type=heads#L129.

Or in the open GitHub in the appendix repository:

https://github.com/jaflaten/master-thesis-appendix/blob/main/src/algorithm/CalculateComputedPriority.java

All classes related to *priority* can be found in the *alien.priority* package (requires CERN account).

https://gitlab.cern.ch/jalien/jalien/-/tree/master/src/main/java/alien/priority.

The source code to the optimizers in JAliEn can be found at this URL (requires CERN account):

`https://gitlab.cern.ch/jalien/jalien/-/tree/master/src/main/java/alien/optimizers`.

Or find them on GitHub in the appendix repository:

`https://github.com/jaflaten/master-thesis-appendix/tree/main/src/optimizers`

Specifically in the *priority* and *sync* sub packages (requires CERN account).

Priority: `https://gitlab.cern.ch/jalien/jalien/-/tree/master/src/main/java/alien/optimizers/priority`

and Sync: `https://gitlab.cern.ch/jalien/jalien/-/tree/master/src/main/java/alien/optimizers/sync`.

A GitHub-repository has been created to host the appendix of this thesis, including the source code for the novel algorithm and optimizers. The repository can be found at `https://github.com/jaflaten/master-thesis-appendix` in the */code* catalogue. The repository contains the following Java files:

- CalculateComputedPriority.java
- CoreCostDto.java
- PriorityDto.java
- PriorityRegister.java
- PTest.java
- QueueProcessingDto.java
- RunningJobs.java
- User.java
- ActiveUserReconciler.java
- InactiveJobHandler.java
- JobAgentUpdater.java
- PriorityRapidUpdater.java
- PriorityReconciliationService.java
- SitequeueReconciler.java

- MasterSubJobReconciler.java
- OldJobRemover.java
- OverwaitingJobHandler.java

In addition to the code for the algorithm and the optimizers there are several files included that were used to run simulations and setup the tests, such as the *PTest.java* file. All code files are available at the following link in the open GitHub-repository:

`https://github.com/jaflaten/master-thesis-appendix/tree/main/src`

Disclaimer: These are individual code files intended solely to showcase the code produced during this thesis. They are not configured for execution out of the box.

Appendix B

Simulations

Plots

A set of simulations executed on different datasets with different input models are visualized as plots on Kaggle using Python. URL: `https://www.kaggle.com/code/jaflaten/notebook3c7187e1af`. Below you will find the input data for the different datasets, including the ones discussed in Chapter 3-5. See the Kaggle link to view all the plots.

The input data

The input data for users vary for each of the values for the aforementioned plots found on the Kaggle URL above. The following tables show the input data for the users in the different datasets.

- ProdUsers1 - Table B.1
- ProdUsers12 - Table B.2
- ProdUsers15 - Table B.3
- ProdUsers16 - Table B.4
- ProdUsers17 - Table B.5
- ProdUsers18 - Table B.6

- ProdUsers19 - Table B.7

	User 974855	**User 1234576**	**User 1234578**	**User 1235889**
User ID	974855	1234576	1234578	1235889
Baseline priority	10.0	20000.0	12000.0	12000.0
Max Parallel Jobs	200000	60000	50000	85000
Userload	0.082065	0.500733	0.18894	0.159882
Max Total Running Time	1.0×10^{12}	1.0×10^{12}	1.0×10^{12}	1.0×10^{14}
Running	16342	30044	9918	13442
Computed Priority	0.0584273	49.9024	115.024	81.2414
Max Total CPU Cost	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}
Total Running Time Last 24h	7.1×10^{9}	2.8×10^{8}	3.4×10^{8}	7.9×10^{8}
Total CPU Cost Last 24h	1.0×10^{4}	5.0×10^{3}	5.0×10^{5}	1.2×10^{7}

Table B.1: User data for 'getProdUsers1()' with formal scientific notation. These values are before any calculations are performed and the *computed-Priority* value in the table is a preset value. A baseline priority of 20000 is the highest of any user in the system.

	User 974855	**User 1234576**	**User 1234578**	**User 1235889**	**User 1235997**
User ID	974855	1234576	1234578	1235889	1235997
Baseline priority	10.0	20000.0	12000.0	12000.0	1.0
Max Parallel Jobs	200000	60000	50000	85000	1000
Userload	0.58171	0.8340667	0.07836	0.7816706	0.7816706
Max Total Running Time	1.0×10^{12}	1.0×10^{12}	1.0×10^{12}	1.0×10^{14}	1.0×10^{7}
Running	0	0	0	0	0
Computed Priority	0.99	0.99	0.99	0.99	0.99
Max Total CPU Cost	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{9}
Total Running Time Last 24h	0	0	0	0	0
Total CPU Cost Last 24h	0.0	0.0	0.0	0.0	0.0

Table B.2: User data for 'getProdUsers12()' with formal scientific notation. These values are before any calculations are performed and the *computed-Priority* value in the table is a preset value. A baseline priority of 20000 is the highest of any user in the system.

The datasets

The CSV files produced by the simulations are available on the following url: `https://www.kaggle.com/datasets/jaflaten/prodsim42`

	User 974855	User 1234576	User 1234578	User 1235889	User 1235997
User ID	974855	1234576	1234578	1235889	1235997
Baseline priority	10.0	20000.0	15000.0	15000.0	1.0
Max Parallel Jobs	200000	60000	50000	85000	1000
Userload	0.58171	0.8340667	0.07836	0.7816706	0.7816706
Max Total Running Time	1.0×10^{12}	1.0×10^{12}	1.0×10^{12}	1.0×10^{14}	1.0×10^{7}
Running	0	0	0	0	0
Computed Priority	0.99	0.99	0.99	0.99	0.99
Max Total CPU Cost	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{9}
Total Running Time Last 24h	0	0	0	0	0
Total CPU Cost Last 24h	0.0	0.0	0.0	0.0	0.0

Table B.3: User data for 'getProdUsers15()' with formal scientific notation. These values are before any calculations are performed and the *computed-Priority* value in the table is a preset value. A baseline priority of 20000 is the highest of any user in the system.

	User 974855	User 1234576	User 1234578	User 1235889	User 1235997
User ID	974855	1234576	1234578	1235889	1235997
Baseline priority	10.0	20000.0	16000.0	16000.0	1.0
Max Parallel Jobs	200000	60000	50000	85000	1000
Userload	0.58171	0.8340667	0.07836	0.7816706	0.7816706
Max Total Running Time	1.0×10^{12}	1.0×10^{12}	1.0×10^{12}	1.0×10^{14}	1.0×10^{7}
Running	0	0	0	0	0
Computed Priority	0.99	0.99	0.99	0.99	0.99
Max Total CPU Cost	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{9}
Total Running Time Last 24h	0	0	0	0	0
Total CPU Cost Last 24h	0.0	0.0	0.0	0.0	0.0

Table B.4: User data for 'getProdUsers16()' with formal scientific notation. These values are before any calculations are performed and the *computed-Priority* value in the table is a preset value. A baseline priority of 20000 is the highest of any user in the system.

	User 974855	User 1234576	User 1234578	User 1235889	User 1235997
User ID	974855	1234576	1234578	1235889	1235997
Baseline priority	10.0	20000.0	17000.0	17000.0	1.0
Max Parallel Jobs	200000	60000	50000	85000	1000
Userload	0.58171	0.8340667	0.07836	0.7816706	0.7816706
Max Total Running Time	1.0×10^{12}	1.0×10^{12}	1.0×10^{12}	1.0×10^{14}	1.0×10^{7}
Running	0	0	0	0	0
Computed Priority	0.99	0.99	0.99	0.99	0.99
Max Total CPU Cost	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{9}
Total Running Time Last 24h	0	0	0	0	0
Total CPU Cost Last 24h	0.0	0.0	0.0	0.0	0.0

Table B.5: User data for 'getProdUsers17()' with formal scientific notation. These values are before any calculations are performed and the *computed-Priority* value in the table is a preset value. A baseline priority of 20000 is the highest of any user in the system.

	User 974855	**User 1234576**	**User 1234578**	**User 1235889**	**User 1235997**
User ID	974855	1234576	1234578	1235889	1235997
Baseline priority	10.0	20000.0	18000.0	18000.0	1.0
Max Parallel Jobs	200000	60000	50000	85000	1000
Userload	0.58171	0.8340667	0.07836	0.7816706	0.7816706
Max Total Running Time	1.0×10^{12}	1.0×10^{12}	1.0×10^{12}	1.0×10^{14}	1.0×10^{7}
Running	0	0	0	0	0
Computed Priority	0.99	0.99	0.99	0.99	0.99
Max Total CPU Cost	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{9}
Total Running Time Last 24h	0	0	0	0	0
Total CPU Cost Last 24h	0.0	0.0	0.0	0.0	0.0

Table B.6: User data for ‘getProdUsers18()‘ with formal scientific notation. These values are before any calculations are performed and the *computed-Priority* value in the table is a preset value. A baseline priority of 20000 is the highest of any user in the system.

	User 974855	**User 1234576**	**User 1234578**	**User 1235889**	**User 1235997**
User ID	974855	1234576	1234578	1235889	1235997
Baseline priority	10.0	20000.0	19000.0	19000.0	1.0
Max Parallel Jobs	200000	60000	50000	85000	1000
Userload	0.58171	0.8340667	0.07836	0.7816706	0.7816706
Max Total Running Time	1.0×10^{12}	1.0×10^{12}	1.0×10^{12}	1.0×10^{14}	1.0×10^{7}
Running	0	0	0	0	0
Computed Priority	0.99	0.99	0.99	0.99	0.99
Max Total CPU Cost	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{14}	1.0×10^{9}
Total Running Time Last 24h	0	0	0	0	0
Total CPU Cost Last 24h	0.0	0.0	0.0	0.0	0.0

Table B.7: User data for ‘getProdUsers19()‘ with formal scientific notation. These values are before any calculations are performed and the *computed-Priority* value in the table is a preset value. A baseline priority of 20000 is the highest of any user in the system.

Appendix C

Database

JAliEn has multiple different central databases, among them are Catalogue, Transfer and TaskQueue. The following tables are from the TaskQueue database. The chosen five tables are the most relevant to the work done in this project. The tables are SITEQUEUES, JOBAGENT, PRIORITY, QUEUE, and QUEUEPROC. The tables are shown with all rows and original field names.

Field	**Type**	**Null**	**Key**	**Default**	**Extra**
siteId	int	NO		0	
DONE_WARN	int	NO		0	
ERROR_EW	int	NO		0	
runload	float	YES			
FORCEMERGE	int	NO		0	
WAITING	int	NO		0	
ERROR_M	int	NO		0	
ERROR_SV	int	NO		0	
FAILED	int	NO		0	
IDLE	int	NO		0	
SAVED	int	NO		0	
A_STAGED	int	NO		0	
ERROR_RE	int	NO		0	
blocked	varchar(20)	NO		locked	
INCORRECT	int	NO		0	

cost	float	YES			
ERROR_A	int	NO		0	
timeblocked	datetime	YES			
MERGING	int	NO		0	
DONE	int	NO		0	
ERROR_I	int	NO		0	
INTERACTIV	int	NO		0	
ERROR_S	int	NO		0	
SAVED_WARN	int	NO		0	
STARTED	int	NO		0	
EXPIRED	int	NO		0	
KILLED	int	NO		0	
ERROR_V	int	NO		0	
TO_STAGE	int	NO		0	
maxqueued	int	NO		0	
INSERTING	int	NO		0	
SPLITTING	int	NO		0	
jdlAgent	text	YES			
queueload	float	NO		0	
status	varchar(25)	NO		new	
OVER_WAITING	int	NO		0	
ERROR_VN	int	NO		0	
ASSIGNED	int	NO		0	
FAULTY	int	NO		0	
ERROR_VT	int	NO		0	
ZOMBIE	int	NO		0	
ERROR_IB	int	NO		0	
SPLIT	int	NO		0	
jdl	mediumtext	YES			
UPDATING	int	NO		0	
ERROR_SPLT	int	NO		0	
RUNNING	int	NO		0	
site	varchar(40)	NO			
statustime	int	NO		0	
ERROR_E	int	NO		0	
maxrunning	int	NO		0	

SAVING	int	NO		0	
STAGING	int	NO		0	
lastRejectionTime	bigint	YES			
lastRejectionReason	varchar(255)	NO		Full match	
ERROR_W	int	NO		0	

Table C.1: SITEQUEUES table structure with all rows and original field names

Field	**Type**	**Null**	**Key**	**Default**	**Extra**
entryId	int	NO	PRI		auto_increment
priority	int	YES	MUL		
noce	varchar(1000)	YES			
fileBroker	tinyint(1)	NO		0	
partition	varchar(1000)	YES			
disk	int	YES			
ttl	int	YES	MUL		
ce	varchar(1000)	YES			
userId	int	NO	MUL		
packages	varchar(1000)	YES			
site	varchar(1000)	YES			
counter	int	NO		0	
price	float	YES	MUL	1	
oldestQueueId	bigint	YES	MUL		
revision	int	YES		0	
cpucores	int	YES		1	
containsAVX	tinyint(1)	YES			
hostname	varchar(100)	YES			
cpuModelName	varchar(1000)	YES			
platforms	varchar(100)	YES			
memory	int	YES			

Table C.2: JOBAGENT table structure with all rows and original field names

Field	Type	Null	Key	Default	Extra
userId	int	NO	PRI		
priority	float	NO		0	
maxparallelJobs	int	NO		20	
unfinishedJobsLast24h	int	NO		0	
userload	float	NO		0	
nominalparallelJobs	int	NO		0	
maxTotalRunningTime	bigint	NO		10000000	
maxUnfinishedJobs	int	NO		1500	
computedpriority	float	NO		0	
maxTotalCpuCost	float	NO		10000000	
totalRunningTimeLast24h	bigint	NO		0	
waiting	int	NO		0	
running	int	NO		0	
totalCpuCostLast24h	float	NO		0	
active	int	YES		0	

Table C.3: PRIORITY table structure with all rows and original field names

Field	Type	Null	Key	Default	Extra
queueId	bigint	NO	PRI		auto_increment
priority	tinyint	YES			
statusId	tinyint	NO	MUL		
submitHostId	int	YES	MUL		
finalPrice	float	YES			
sent	int	YES			
split	bigint	YES	MUL		
nodeId	int	YES	MUL		
execHostId	int	YES	MUL		
mtime	timestamp	NO		CURRENT_TIMESTAMP	DEFAULT_GENERATED on update
agentId	int	YES	MUL		
price	float	YES			
finished	int	YES			
masterjob	int	YES		0	
splitting	int	YES			
notifyId	int	YES	MUL		
optimized	int	YES		0	
commandId	int	YES	MUL		
error	int	YES			
resubmission	int	NO		0	
received	int	YES			
validate	int	YES			
merging	varchar(64)	YES			
userId	int	YES	MUL		
chargeStatus	varchar(20)	YES			
siteId	int	NO	MUL	1	
started	int	YES			
expires	int	YES			
remoteTimeout	int	YES			
cpucores	int	YES		1	
killreason	int	YES			

Table C.4: QUEUE table structure with all rows and original field names

Field	Type	Null	Key	Default	Extra
queueId	bigint	NO	PRI		
lastupdate	timestamp	NO		CURRENT_TIMESTAMP	DEFAULT_GENERATED on update
maxrsize	float	YES			
cputime	int	YES			
ncpu	int	YES			
batchid	varchar(255)	YES			
cost	float	YES			
cpufamily	int	YES			
cpu	float	YES			
rsize	int	YES			
spyurl	varchar(64)	YES			
runtime	varchar(20)	YES			
mem	float	YES			
si2k	float	YES			
cpuspeed	int	YES			
vsize	int	YES			
runtimes	int	YES			
procinfotime	int	YES			
maxvsize	float	YES			
killreason	int	YES			

Table C.5: QUEUEPROC table structure with all rows and original field names

Appendix D

Diagrams

Several drawings and diagrams have been created during the work on this thesis. Some of the figures are too large to fit on a single page and are therefore uploaded to a public GitHub-repository to host the appendix of this thesis. The repository can be found at `https://github.com/jaflaten/master-thesis-appendix`. Several of the figures are of interest to understand JAliEn, but not all would fit in the written work, below you can find links to some of the most prominent ones, but all figures are available in the */images* catalogue in the GitHub-repository. The repository contains the following figures:

- JAliEn architecture diagram (detailed)
- JAliEn architecture diagram (simplified)
- Central and site services in JAliEn
- Job state flow diagram
- Final and error states in JAliEn
- Simplified flow from user point of view
- To zombie state optimizer flow
- Job submission by user flow
- Legacy bookkeeping service flow

- Design science research
- Grid Protocol Architecture vs Internet Protocol Architecture
- Multilevel queue scheduling
- First come, first served
- Shorted Job First
- Shortest Remaining Time First
- Round Robin
- Priority Scheduling
- XRootD overview

JAliEn flow state diagram

In JAliEn there are several different states (status) a job can be in. See Table D.1 below for the Some state are a part of the normal job lifecycle, while others are error or other final state that a job might end up in. The state flow diagram follows a job from submission to the end. A successful path could be straight downward, while erroneous paths or a split job could take a different route. The state diagram is available at the following link:

`https://github.com/jaflaten/master-thesis-appendix/blob/main/images/state%20flow%20diagram%20of%20job%20states%20in%20JAliEn.png`

The final and error states can be seen here:

`https://github.com/jaflaten/master-thesis-appendix/blob/main/images/final%20and%20error%20states%20in%20JAliEn.png`

The following is a simplified flow where a user submits a JDL and the resource usage is submitted and *computedPriority* is calculated and updated.

`https://github.com/jaflaten/master-thesis-appendix/blob/main/images/simplified%20flow%20from%20user%20pov.png`

The next figure is a scenario that does not exist, but is something that was discussed as a possible feature. When a job is moved to a ZOMBIE state it is not remembered what the previous state was. So there is no possibility to recover if a new heartbeat is received. This would require some rewriting of

the system to remember previous states, these would also have to be stored in the database.

https://github.com/jaflaten/master-thesis-appendix/blob/main/images/to%20zombie%20state%20optimizer%20flow.png

This figure is a overview of a user submitting a job in the legacy system before the novel algorithm.

https://github.com/jaflaten/master-thesis-appendix/blob/main/images/job%20submission%20by%20user.png

Architecture

Multiple architecture diagrams have been created during the course of this work. The first one is simpled higher level overview of the JAliEn system. The second one is a more detailed view of the system.

Overview: https://github.com/jaflaten/master-thesis-appendix/blob/main/images/simplified%20jalien%20architecture.png

Detailed: https://github.com/jaflaten/master-thesis-appendix/blob/main/images/JAliEn%20architecture.png

This figure displays a grid site and some details about the services that are running on the site.

https://github.com/jaflaten/master-thesis-appendix/blob/main/images/central%20and%20site%20services.png

Status	**Explanation**
INSERTING	Job being inserted into the system
ERROR_I	Error while inserting job
WAITING	Job is waiting in queue after insertion
ASSIGNED	Assigned to node for execution
STARTING	Execution of job started
ERROR_EW	Waiting for over 7 days without being assigned
ERROR_A	Error immediately after assigning
ERROR_IB	Job setup failed after assigning
KILLED	Job is killed while ASSIGNED, STARTING or RUNNING
RUNNING	Job is running and being executed
ERROR_E	Non-zero exit code, such as exceeding disk, time, or memory limits
ERROR_YT	Timeout error during validation
ERROR_VN	Error starting validation
ERROR_V	Validation not successful
ERROR_S	File not available during saving
ERROR_SV	Problem saving file
SAVING	Saving job
DONE	Job completed with all replica files
DONE_WARN	Job completed without all files
ZOMBIE	No heartbeat for over an hour
EXPIRED	Inactive for two hours
SPLITTING	Splitting a master job
SPLIT	Successfully split master job
ERROR_SPLT	Error when splitting job

Table D.1: Job Status Definitions

Bibliography

[1] Miron Livny et al. "Mechanisms for High Throughput Computing." In: 1997. URL: https://api.semanticscholar.org/CorpusID:19031577.

[2] K. Aamodt et al. "The ALICE experiment at the CERN LHC." In: *JINST* 3 (2008), S08002. DOI: 10.1088/1748-0221/3/08/S08002.

[3] Martinez Pedreira, M., Grigoras, C., and Yurchenko, V. "JAliEn: the new ALICE high-performance and high-scalability Grid framework." In: *EPJ Web Conf.* 214 (2019), p. 03037. DOI: 10.1051/epjconf/201921403037. URL: https://doi.org/10.1051/epjconf/201921403037.

[4] ALICE Collaboration. *Technical Design Report for the Upgrade of the ALICE Readout & Trigger System*. Tech. rep. ALICE-TDR-019. Chapters 4 and 5, pp. 21-48. Accessed: 2024-11-05. CERN, 2014. URL: https://cds.cern.ch/record/2011297/files/ALICE-TDR-019.pdf.

[5] Weisz, Sergiu and Ferrer, Marta Bertran. "Adding multi-core support to the ALICE Grid Middleware." In: *Journal of Physics: Conference Series*. Vol. 2438. 1. IOP Publishing. 2023, p. 012009.

[6] Edsger W. Dijkstra. "The structure of the "THE"-multiprogramming system." In: *Proceedings of the First ACM Symposium on Operating System Principles*. SOSP '67. New York, NY, USA: Association for Computing Machinery, 1967, pp. 10.1–10.6. ISBN: 9781450373708. DOI: 10.1145/800001.811672. URL: https://doi.org/10.1145/800001.811672.

[7] Richard Conway, William L. Maxwell, and Louis W. Miller. "Theory of scheduling." In: 1967. URL: https://api.semanticscholar.org/CorpusID:60926329.

[8] Jack B. Dennis and Earl C. Van Horn. "Programming semantics for multiprogrammed computations." In: *Commun. ACM* 9.3 (Mar. 1966),

pp. 143–155. ISSN: 0001-0782. DOI: 10.1145/365230.365252. URL: https://doi.org/10.1145/365230.365252.

[9] The Linux Kernel Documentation Team. *CFS Scheduler Design*. Accessed: 2024-11-19. 2024. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[10] The Linux Kernel Documentation Team. *EEVDF Scheduler*. Accessed: 2024-11-19. 2024. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-eevdf.html.

[11] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 10th ed. See pages 200–250. Hoboken, NJ: John Wiley & Sons, 2018. ISBN: 978-1119456339.

[12] E. A. Huerta et al. "Supporting High-Performance and High-Throughput Computing for Experimental Science." In: *Computing and Software for Big Science* 3.1 (Feb. 2019). ISSN: 2510-2044. DOI: 10.1007/s41781-019-0022-7. URL: http://dx.doi.org/10.1007/s41781-019-0022-7.

[13] J Balcas et al. "MonALISA, an agent-based monitoring and control system for the LHC experiments." In: *Journal of Physics: Conference Series* 898.9 (Oct. 2017), p. 092055. DOI: 10.1088/1742-6596/898/9/092055. URL: https://dx.doi.org/10.1088/1742-6596/898/9/092055.

[14] Mihael Hategan-Marandiuc et al. "PSI/J: A Portable Interface for Submitting, Monitoring, and Managing Jobs." In: *2023 IEEE 19th International Conference on e-Science (e-Science)*. IEEE, Oct. 2023, pp. 1–10. DOI: 10.1109/e-science58273.2023.10254912. URL: http://dx.doi.org/10.1109/e-Science58273.2023.10254912.

[15] Igor Sfiligoi, Thomas DeFanti, and Frank Würthwein. "Auto-scaling HTCondor pools using Kubernetes compute resources." In: *Practice and Experience in Advanced Research Computing*. PEARC '22. ACM, July 2022, pp. 1–4. DOI: 10.1145/3491418.3535123. URL: http://dx.doi.org/10.1145/3491418.3535123.

[16] C Hollowell et al. "Mixing HTC and HPC Workloads with HTCondor and Slurm." In: *Journal of Physics: Conference Series* 898.8 (Oct. 2017), p. 082014. DOI: 10.1088/1742-6596/898/8/082014. URL: https://dx.doi.org/10.1088/1742-6596/898/8/082014.

[17] SchedMD. *SLURM Overview*. https://slurm.schedmd.com/overview.html. Accessed: 2024-11-01. 2024.

[18] Wikipedia contributors. *SLURM Workload Manager*. https://en.wikipedia.org/w/index.php?title=Slurm_Workload_Manager&oldid=1246246065. Accessed: 2024-11-01. 2024.

[19] Ian Foster, Carl Kesselman, and Steven Tuecke. *The Anatomy of the Grid - Enabling Scalable Virtual Organizations*. 2001. arXiv: cs/0103025 [cs.AR]. URL: https://arxiv.org/abs/cs/0103025.

[20] Henri Casanova. "Distributed computing research issues in grid computing." In: *SIGACT News* 33.3 (Sept. 2002), pp. 50–70. ISSN: 0163-5700. DOI: 10.1145/582475.582486. URL: https://doi-org.galanga.hvl.no/10.1145/582475.582486.

[21] John Strassner et al. *Network Policy and Services: A Report of a Workshop on Middleware*. RFC 2768. Feb. 2000. DOI: 10.17487/RFC2768. URL: https://www.rfc-editor.org/info/rfc2768.

[22] F. Gregoretti et al. "MGF: A grid-enabled MPI library." In: *Future Generation Computer Systems* 24.2 (2008), pp. 158–165. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2007.03.009. URL: https://www.sciencedirect.com/science/article/pii/S0167739X07000519.

[23] Edgar Gabriel et al. "Distributed Computing in a Heterogeneous Computing Environment." In: (Sept. 1998).

[24] Fred Baker. *Requirements for IP Version 4 Routers*. RFC 1812. June 1995. DOI: 10.17487/RFC1812. URL: https://www.rfc-editor.org/info/rfc1812.

[25] Y Butler, Steven Tuecke, and John Volmer. "Design and Deployment of a National-Scale Authentication Infrastructure." In: (Aug. 2000).

[26] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[27] James Sermersheim. *Lightweight Directory Access Protocol (LDAP): The Protocol*. RFC 4511. June 2006. DOI: 10.17487/RFC4511. URL: https://www.rfc-editor.org/info/rfc4511.

[28] John Bresnahan et al. "Gridftp pipelining." In: (Jan. 2007).

[29] Globus. *Globus Connect*. Accessed: 2024-10-26. 2024. URL: https://www.globus.org/globus-connect.

[30] CERN. *Fast Data Transfer (FDT)*. Accessed: 2024-10-07. 2024. URL: http://monalisa.cern.ch/FDT/.

[31] Alvise Dorigo et al. “XROOTD - A highly scalable architecture for data access.” In: *WSEAS Transactions on Computers* 4 (Apr. 2005), pp. 348–353.

[32] Alice Florența Șuiu et al. “File Spooler and Copy System for Fast Data Transfer.” In: *7th Conference on the Engineering of Computer Based Systems*. ECBS 2021. Novi Sad, Serbia: Association for Computing Machinery, 2021. ISBN: 9781450390576. DOI: 10.1145/3459960.3461559. URL: https://doi.org/10.1145/3459960.3461559.

[33] H. K. Huang et al. “Data grid for large-scale medical image archive and analysis.” In: *Proceedings of the 13th Annual ACM International Conference on Multimedia*. MULTIMEDIA ’05. Hilton, Singapore: Association for Computing Machinery, 2005, pp. 1005–1013. ISBN: 1595930442. DOI: 10.1145/1101149.1101357. URL: https://doi-org.galanga.hvl.no/10.1145/1101149.1101357.

[34] GeeksforGeeks. *Middleware in Grid Computing*. Accessed: 2024-11-04. 2024. URL: https://www.geeksforgeeks.org/middleware-in-grid-computing/.

[35] CERN. *Grid Software, Middleware and Hardware*. Accessed: 2024-11-04. 2024. URL: https://home.cern/science/computing/grid-software-middleware-hardware.

[36] A. G. Grigoras et al. “JAliEn – A new interface between the AliEn jobs and the central services.” In: *Journal of Physics: Conference Series* 523 (2014). Published under licence by IOP Publishing Ltd, p. 012010. DOI: 10.1088/1742-6596/523/1/012010.

[37] J. Blomer et al. “Distributing LHC application software and conditions databases using the CernVM file system.” In: *Journal of Physics: Conference Series* 331 (2011), p. 042003. DOI: 10.1088/1742-6596/331/4/042003.

[38] Costin Grigoras. *ALICE Day, UPB 2024*. Accessed: 2024-11-11. URL: https://docs.google.com/presentation/d/1trFR-z4LRJvywY73EaeXKoOzYnPBb7O-D4LOOMhpUrU/edit#slide=id.g30925f569e6_15_30.

[39] Costin Grigoras et al. *JAliEn - T1/T2 workshop - 2022-09-26*. Accessed: 2024-11-11. URL: https://docs.google.com/presentation/d/1p8AmNZPVhzvM02buFWIXL83nfivdggHym-rOe-qsG6s/edit#slide=id.gfeea2917d9_0_50.

[40] M. Cooper, P. Hesse, et al. “X.509 Public Key Infrastructure.” In: *IETF RFC4158* (2005).

[41] Miguel Martinez Pedreira et al. "The security model of the ALICE next generation Grid framework." In: *EPJ Web Conf.* 214 (2019), p. 03042. DOI: 10.1051/epjconf/201921403042. URL: https://cds.cern.ch/record/2701499.

[42] MySQL Developer Documentation. *Glossary*. Accessed: 2024-11-04. 2024. URL: https://dev.mysql.com/doc/refman/8.4/en/glossary.html.

[43] MySQL Developer Documentation. *Glossary: Referential Integrity*. Accessed: 2024-11-04. 2024. URL: https://dev.mysql.com/doc/refman/8.4/en/glossary.html#glos_referential_integrity.

[44] MySQL Developer Documentation. *Glossary: ACID*. Accessed: 2024-11-04. 2024. URL: https://dev.mysql.com/doc/refman/8.4/en/glossary.html#glos_acid.

[45] MySQL Developer Documentation. *Glossary: Transaction*. Accessed: 2024-11-04. 2024. URL: https://dev.mysql.com/doc/refman/8.4/en/glossary.html#glos_transaction.

[46] MySQL Developer Documentation. *InnoDB Introduction*. Accessed: 2024-11-04. 2024. URL: https://dev.mysql.com/doc/refman/8.4/en/innodb-introduction.html.

[47] MySQL Developer Documentation. *InnoDB Transaction Isolation Levels*. Accessed: 2024-11-04. 2024. URL: https://dev.mysql.com/doc/refman/8.4/en/innodb-transaction-isolation-levels.html.

[48] Oracle Documentation Team. *Data Concurrency and Consistency*. Accessed: 2024-11-19. 2024. URL: https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/data-concurrency-and-consistency.html#GUID-7AD41DFA-04E5-4738-B744-C4407170411C.

[49] MySQL Developer Documentation. *Glossary: Concurrency*. Accessed: 2024-11-05. 2024. URL: https://dev.mysql.com/doc/refman/8.4/en/glossary.html#glos_concurrency.

[50] Wikipedia contributors. *Multiversion Concurrency Control*. Accessed: 2024-11-19. 2024. URL: https://en.wikipedia.org/w/index.php?title=Multiversion_concurrency_control&oldid=1256423603.

[51] MySQL Developer Documentation. *InnoDB Consistent Read*. Accessed: 2024-11-19. 2024. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-consistent-read.html.

[52] MySQL Developer Documentation. *Glossary: Optimistic Concurrency*. Accessed: 2024-11-05. 2024. URL: https://dev.mysql.com/doc/refman/8.4/en/glossary.html#glos_optimistic.

[53] IBM Documentation. *CAP Theorem*. Accessed: 2024-11-04. 2024. URL: https://www.ibm.com/topics/cap-theorem.

[54] Eric Brewer. "CAP twelve years later: How the "rules" have changed." In: *Computer* 45.2 (2012), pp. 23–29. DOI: 10.1109/MC.2012.37.

[55] Ramtin Jabbari et al. "What is DevOps? A Systematic Mapping Study on Definitions and Practices." In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. XP '16 Workshops. Edinburgh, Scotland, UK: Association for Computing Machinery, 2016. ISBN: 9781450341349. DOI: 10.1145/2962695.2962707. URL: https://doi.org/10.1145/2962695.2962707.

[56] Alan R. Hevner. "A Three Cycle View of Design Science Research." In: *Scandinavian Journal of Information Systems* 19.2 (2007).

[57] Alan R. Hevner et al. "Design Science in Information Systems Research." In: *MIS Quarterly* 28.1 (2004), pp. 75–105.

[58] Herbert Alexander Simon. *The Sciences of the Artificial*. English. MIT Press, 1996. ISBN: 0262193744.

[59] Oracle Corporation. *Java Platform, Standard Edition 11 Documentation*. 2024. URL: https://docs.oracle.com/en/java/javase/11/.

[60] Gradle Inc. *Gradle Build Tool*. 2024. URL: https://gradle.org/.

[61] JetBrains. *IntelliJ IDEA - The Leading Java and Kotlin IDE*. 2024. URL: https://www.jetbrains.com/idea/.

[62] JetBrains. *DataGrip - IDE for Databases and SQL*. 2024. URL: https://www.jetbrains.com/datagrip/.

[63] Excalidraw. *Excalidraw - Hand-drawn look diagrams*. 2024. URL: https://excalidraw.com/.

[64] Huajie Xu. "Github Copilot - A Groundbreaking Code Autocomplete Tool." In: Dec. 2023. DOI: 10.13140/RG.2.2.29962.24002.

[65] Python Software Foundation. *Python 3.9 Documentation*. 2024. URL: https://docs.python.org/3/.

[66] Kaggle Inc. *Kaggle - Your Home for Data Science*. 2024. URL: https://www.kaggle.com/.

[67] Kirrily "Skud" Robert. *Perl documentation, intro*. https://perldoc.perl.org/perlintro. (Visited on Mar. 30, 2024).

[68] MySQL Developer Documentation. *SQL Prepared Statements*. Accessed: 2024-09-26. 2024. URL: https://dev.mysql.com/doc/refman/8.4/en/sql-prepared-statements.html.

[69] CERN developers et al. *AliEn - original algorithm for computedPriority*. https://gitlab.cern.ch/jflaten/alien-old/. (Visited on Mar. 30, 2024).

[70] Oracle et al. *JavaDoc - Class Math*. https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html. (Visited on Apr. 2, 2024).

[71] Alexandar Pantaleev and Atanas Rountev. "Identifying Data Transfer Objects in EJB Applications." In: *Fifth International Workshop on Dynamic Analysis (WODA '07)*. 2007, pp. 5–5. DOI: 10.1109/WODA.2007.6.

[72] MySQL Developer Documentation. *InnoDB Transaction Isolation Levels*. Accessed: 2024-09-12. 2024. URL: https://dev.mysql.com/doc/refman/8.4/en/innodb-transaction-isolation-levels.html.

[73] Oracle et al. *JavaDoc: Class ConcurrentHashMap ¡ K,V¿*. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html. (Visited on Mar. 28, 2024).

[74] R. L. Harrison. "Introduction To Monte Carlo Simulation." In: *AIP Conf Proc* 1204 (2010), pp. 17–21. DOI: 10.1063/1.3295638. URL: https://doi.org/10.1063/1.3295638.